REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
29/12/2006	Final Report	March 2003-December 2006

4. TITLE AND SUBTITLE
Adaptive Multilevel Middleware for Object Systems

5a. CONTRACT NUMBER
NBCHC030119

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dr. Richard E. Schantz
Dr. Joseph P. Loyall
Dr. Kurt Rohloff
Dr. Jianming Ye
Mr. Matthew Gillen
Mr. Paul Rubel
Mr. Prakash Manghwani
Mr. Yarom Gabay
Dr. Priya Narasimhan
Mr. Aaron Paulos
Dr. Aniruddha Gokhale
Mr. Jaiganesh Balasubramanian

5d. PROJECT NUMBER
Q53000

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BBN Technologies, 10 Moulton Street, Cambridge, MA 02138
Vanderbilt University, Nashville TN
Carnegie-Mellon University, Pittsburgh, PA

8. PERFORMING ORGANIZATION REPORT NUMBER
13029

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 Fairfax Drive
Arlington, VA 22203-1714
U.S. Department of the Interior
National Business Center, Acquisition & Property Management Division
PO Box 12924
Ft. Huachuca, AZ 85670

10. SPONSOR/MONITOR'S ACRONYM(S)
DARPA/IXO, DOI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; Distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The ARMS program researched and developed state-of-the-art technologies in dynamic quality of service and resource management, and applied them to the challenges of modern total ship Naval computing platforms. The program was organized in two phases, with Phase 1 concentrating on the research of underlying multi-layered resource management concepts and the design and prototyping of an integrated multi-layer resource management (MLRM) capability. Phase 2 then concentrated on additional research in areas building upon this MLRM capability, to develop significantly greater capabilities in the areas of resource and QoS management algorithms and MLRM fault tolerance, and to transition ARMS technologies to the Naval Program of Record (PoR). This report describes the research, development, and transition activities and results of the BBN Technologies project within the ARMS program.

INSTRUCTIONS FOR COMPLETING SF 298

15. SUBJECT TERMS Service and Resource Management Computing Platforms					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT U	18. NUMBER OF PAGES 196	19a. NAME OF RESPONSIBLE PERSON Samantha Smithwick
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPONE NUMBER (*Include area code*) 703-284-1335

ADAPTIVE MULTILEVEL MIDDLEWARE FOR OBJECT SYSTEMS

Final Report

December 2006

Contract Number	NBCHC030119
Period Covered	March 2003 to December 2006
For BBN Technologies	Dr. Richard E. Schantz Dr. Joseph P. Loyall Dr. Kurt Rohloff Dr. Jianming Ye Mr. Matthew Gillen Mr. Paul Rubel Mr. Prakash Manghwani Mr. Yarom Gabay
For Carnegie Mellon University	Dr. Priya Narasimhan Mr. Aaron Paulos
For Vanderbilt University	Dr. Aniruddha Gokhale Mr. Jaiganesh Balasubramanian

Prepared for:
Dr. Joseph Cross
DARPA / IXO
3701 Fairfax Drive
Arlington, VA 22203-1714

Prepared by:
BBNT Solutions LLC.
10 Moulton Street
Cambridge, MA 02138

BBN TECHNOLOGIES

This page is blank intentionally.

CONTENTS

Figures

Tables

1. Executive Summary and Introduction

The goals of the ARMS program were to research and develop state-of-the-art technologies providing and supporting dynamic Quality of service and resource management, and to apply them to the challenges of developing modern total ship Naval computing platforms. The program was organized in two phases, with Phase 1 concentrating on the concepts underlying research in multi-layered, dynamic resource management and the design and prototyping of an integrated multi-layer resource management (MLRM) capability. Phase 2 then concentrated on additional research in areas building upon this MLRM capability, to develop significantly greater capabilities in the areas of resource and QoS management algorithms and MLRM fault tolerance, and to transition ARMS technologies to the Navy Program of Record (PoR).

The ARMS program prime contractors worked on different aspects of the ARMS technologies, with cooperative R&D on some aspects between projects. Each phase of the ARMS program had Gate Metrics to meet, divided into separate, but sometimes related individual Gate Tests (GTs). The gate tests were program milestones quantitatively confirming that the ARMS technologies provide significant improvement over the baseline current state of the practice and were applicable to the requirements of the PoR. .

In Phase 1, the BBN team was instrumental in the design of the MLRM architecture, adapting it to the specific context of the Navy PoR, development of an integrated MLRM prototype, and passing of the performance oriented gate tests to establish operational viability of the concept. Specifically, in addition to initiating many of the driving architectural concepts, the BBN team provided application and end-to-end application string management architectural components; developed a common system service for collecting and disseminating resource status information system wide; provided a standards based software platform for easily linking and connecting the various MLRM software components; led system integration, test and evaluation activities; and introduced and supported a commonly accessible testbed facility to organize and significantly improve multi-technology developer (TD) integration and testing activities. Together with other program participants, we were successful in developing a prototype MLRM subsystem sufficient to demonstrate and measure dynamic resource management principles operating against simulated PoR workloads, and effectively managing a wide variety of alternative configurations, managing application overload while maximizing resources applied to high priority tasks, and recovering from large scale failures. These prototype capabilities were evaluated against pre-established program metrics defined by the Phase 1 gate tests. They showed sufficient maturity and continued potential to provide the intended risk reduction attributes for developing similar operational surface ship capabilities and to warrant an ARMS Phase 2. Our Phase 1 efforts are discussed in Section 2.

During Phase 2 we built upon the accomplishments in Phase 1 to extend the technology base more thoroughly into and across the space of issues underlying multi-layer dynamic resource management. The BBN team conducted R&D in rapid recovery fault tolerance mechanisms and mission-based dynamic resource management algorithms; designed, prototyped and integrated a real-time Fault-Tolerant (FT) functionality applicable to the requirements of an MLRM, developed and transitioned to the Navy PoR a highly scalable and high performance Node Failure Detection capability, and developed Dynamic Resource Management designs, algorithms and methods appropriate for and evaluated against a proxy for the ongoing design of the PoR system. The BBN team led the efforts to successfully pass Gate Test 3 and contributed significantly to the successful completion of the Phase 2 Gate Tests 1 and 4. Our Phase 2 activities are discussed in depth in Sections 3, 0 and 5.

The first major part of our effort during ARMS Phase 2 was spent in researching advanced fault tolerance technologies and enhancing the previously developed non-Fault Tolerant MLRM to make it fault tolerant. Fault tolerance (FT) is a crucial design consideration for mission-critical distributed real-time and embedded (DRE) infrastructure and services, such as the MLRM. DRE systems such as the MLRM combine the real-time characteristics of embedded platforms with the dynamic characteristics of distributed platforms. However, many of the characteristics of these systems, such as heterogeneity, strict timing requirements, scalability, and non-client-server application interactions, prove challenging to implementing a fault-tolerance solution with the techniques and technology that existed at the beginning of ARMS. In order to make the MLRM fault tolerant, we had to design and implement several innovative advancements to the state of the art in fault tolerance. These advancements included (a) enabling the cooperative use of group and non-group communications, and as a result improving efficiency and scalability; (b) developing fault tolerance support for CORBA components and their peer-to-peer calling patterns; (c) developing dynamic deployment of CORBA components; and (d) supporting multiple languages (C++ and Java); among other advances. We describe our development process, the fault tolerance advancements that we made, and how this system was used to pass the ARMS Gate Test 3 in Section 2.

The second major aspect of our effort in ARMS Phase 2 was the design and development of a highly scalable, adaptive, multi-layered node failure detection (NFD) capability. By their nature, mission critical applications often require constant availability. Since no hardware is immune to failures, either from normal wear and tear or from battle damage, these mission critical applications need a NFD capability to provide support for activating backups or backup plans in the case of failures The team at BBN developed a proof-of-concept implementation of a software-based node-failure detection capability, which was both fault-tolerant and highly scalable, while at the same time ensuring that the solution maintained a very low-overhead footprint, in line with the requirements of the PoR. This NFD capability applied and combined our earlier R&D results from the fault tolerance and multi-layered design aspects of the ARMS program, against the set of operational requirements from the PoR for it to be a potential transition candidate. BBN also performed extensive tests of the resulting implementation to demonstrate that all of the PoR's requirements were satisfied. The result of this activity was successful in terms of both advancing the state of the art and in providing the PoR with a drop-in technology to fill their node-detection technology gap. In Section 4, we discuss the design and implementation of the NFD system, and present the results of experiments measuring, evaluating and comparing the R&D version of the NFD against an earlier, non-scaleable, non-fault tolerant version. We also give an account of interactions with the PoR to transition the NFD into use in the PoR environment.

The third major aspect of our Phase 2 effort was the development of dynamic resource management (DRM) algorithms for the Multi-Layer Resource Management system. In the ARMS/MLRM design, we established that system behavior could be decomposed into various missions and every mission in the system could be decomposed into possibly repeated sub-missions called strings. For our dynamic resource management efforts, we developed a set of utility functions as real-time measures of system performance. We used our utility functions to guide the design of a hierarchical resource management system based on a string-mission-system decomposition of system behavior. As we were designing the control system, we developed Matlab/Simulink simulations of the ARMS/MLRM system to examine the benefits of the various resource control algorithms in the warfighter domain context. We used measures of effectiveness taken from the concept of operations for Gate Test 4 (GT4 CONOPS) which was evaluating a simplified variant of dynamic response. We used these simulations to select a set of algorithms for managing the control hierarchy. That design was implemented in the GT4 testbed and extended the simplified GT4 results using a much more detailed and realistic set of system and workload assumptions. Using our algorithms, we were able to achieve an order of magnitude improvement in DRM performance as measured by the GT4 warfighter value metrics. The DRM aspects of Phase 2 efforts are discussed in Section 5.

1.1 Programmatic Data

The ARMS program had 2 phases. The BBN project, "Adaptive Multilevel Middleware for Object Systems" was one of a number of simultaneous and cooperating projects in the ARMS program, and spanned the two phases.

Phase 1 of the Adaptive Multilevel Middleware for Object Systems project ran from November 2003 through March 2005. The Principal Investigator at BBN for the ARMS Phase 1 effort was Dr. Richard Schantz. Vanderbilt University was a subcontractor on phase 1.

Phase 2 of the Adaptive and Reflective Middleware Systems (ARMS) project ran from June, 2005 through December, 2006. The Principal Investigator for the ARMS project was Dr. Richard Schantz and Dr. Joseph P. Loyall was a co-PI. Carnegie Mellon and Vanderbilt Universities were subcontractors on phase 2 of the project.

1.2 Goals of the project

The main goals of this project were to research and develop state-of-the-art technologies in dynamic Quality of service and resource management, and to apply them to the challenges of designing, constructing and fielding modern total ship Navy computing platforms. This will enable the paradigm shift to a total ship computing approach for the computing infrastructure, which requires a more sophisticated and dynamic level of resource management than anything available today. It will be common to all shipboard application elements and manage the collection of resources available on a shared, total ship basis, not just on an individual subsystem basis, reconfiguring to meet evolving and changing requirements, demands, and priorities. There are two sub-goals for this project that derive from the main goal:

- An agile, multi-layer approach to managing resources, where the higher layers set the appropriate policy on a more global (ship) basis, while lower layers monitor and react rapidly to maintain those policies and maximizing derived computational value by regulating local subsystem behavior.
- An adaptive resource management strategy where changes in mission requirements, load, or available resources can lead to rapid reconfiguration with appropriate tradeoffs among the managed properties.

While each of these sub-goals is a significant challenge in its own right, their solution is tightly intertwined. The capability we envision would become part of the standard off-the-shelf middleware interposed between the application subsystems and integrated with lower level common off-the-shelf (COTS) infrastructure elements. It would serve to provision, configure, monitor and adapt the elements requiring coordinated or controlled actions to achieve the appropriate end-to-end QoS results over a shared resource base. It would automatically select the appropriate real-time property management discipline for the current configuration, regulating other aspects as a side effect. It would scale to the size anticipated for larger versions of PoR class platforms and have a reaction/reconfiguration cycle time commensurate with the requirements from the current complement of sensor and weapons platforms anticipated for the PoR. The middleware resource management algorithms and mechanisms will be parameterized and easily replaceable, to allow additional strategies to be inserted as the ship's configuration and the knowledge of how to construct and run these more dynamic software capabilities evolve over time.

At a more specific granularity, the research goals of the BBN team's project in ARMS Phase 1 were:

- Establish a multi-layer architecture, supporting dynamic resource decision making, suitable for the PoR total ship computing context. Investigate and instantiate an application-centric management function with dynamic properties for end-to-end QoS and resource management. With other ARMS researchers, develop a working prototype for the dynamic multi-layer resource management capability.
- Investigate and instantiate a multi-layer resource status service for the shipboard environment that is used to assess current conditions and drive resource reallocation decisions.
- Provide a standards-based component infrastructure on which to build the dynamic resource management capability to demonstrate the feasibility of using an open COTS base capability for shipboard real-time computing.
- Test, measure, evaluate and iterate on the proposed solution to ensure its effectiveness for timely dynamic resource management under changing conditions and in processing typical PoR workloads and missions.

Phase 2 of ARMS augmented the research agenda established and accomplished to a concept enabling degree during phase 1, with the following goals:

- Research new fault tolerance technologies suitable for the dynamic and distributed real-time characteristics of the shipboard infrastructure environment.
- Develop and demonstrate a fault tolerant MLRM in keeping within constraints and requirements from the PoR components of similar functionality.
- Develop advanced resource management strategies and algorithms that incorporate mission priorities and effective usage of resources.
- Lead efforts to successfully pass the Phase 2 Gate Test 3 program evaluation milestone.
- With other researchers, contribute to successfully pass the Phase 2 Gate Test 1 and 4 program evaluation milestones.

1.3 Comparison with Current Technology

There is no current capability technology available for shipboard dynamic resource management. Currently available designs and components use static resource management, with any dynamic decision making being employed in an ad hoc, case at a time manner. Instantiating the dynamic, multi-layer capability proposed here would effect a revolutionary change in the approach toward building more responsive systems over shared resources for PoR types of systems on a common platform. Developing the concepts and designs for such a manageable, shared infrastructure, and demonstrating the feasibility of constructing it to meet the demanding requirements of a PoR system, represents the major change from current practice.

1.4 Organization of This Report

This report is organized into five main parts.

- Section 1 introduces the report with an executive summary. Section 1.1 discusses programmatic data. The goals of the project are discussed in Section 1.2. A description of the technological basis upon which the project started is given in Section1.3, and the organization of the report is described in Section 4.

- Section 2 describes the efforts associated with the program during Phase 1. An overview of the ARMS architecture is given in Section 2.1 Section 2.2 discusses our development activities. Testing and laboratory support are discussed in Sections 2.3 and 2.4 respectively.

- Section 3 describes our development process and demonstrates how this was used to pass Gate Test 3. Section 3.2 discusses the Gate Test 3 results. Fault Tolerant research results are discussed in Section 3.3.

- Section 4 discusses the NFD system. We present the results of various experiments, and compare results with similar experiments using an earlier, less capable NFD prototype. We also give an account of transition-related interactions with the PoR in this section. Section 4 discusses the program's NFD requirements. Section 4.2 discusses the design and implementation of the NFD system. Sections 4.3 and 4.4 describe the NFD technology transition activities. Section 4.5 discusses research into an adaptive NFD capability.

- Section 5 discusses the DRM aspects of the ARMS Phase 2 effort. In Section 5.1 we describe the utility functions developed to measure system performance. In Section 5.2 we describe the initial design of a hierarchical resource management system capability. We refined and tested this initial design with the aid of models of the MLRM system in Matlab/Simulink. These efforts are described in Section 5.3. Section 5.4 describes experiments in using and evaluating our refined DRM technology as part of extending the ARMS Gate Test 4 dynamic response evaluation effort beyond its simplified basics.

- Section 6 contains chronological reviews of ARMS publications and activities.

2. ARMS Phase 1 Development

The main goal of ARMS Phase 1 was to design and prototype a runtime Quality of Service (QoS) management capability appropriate for a common, shared, shipboard network and host platform. This would enable a paradigm shift to a total ship computing approach for the computing infrastructure, requiring a more sophisticated level of QoS management than anything available at the start of the ARMS program. It will be common to all shipboard application elements and manage the collection of resources available on a shared, total ship basis, not just on an individual subsystem basis. There were two sub-goals for the Phase 1 effort of this project derived from the main goal:

1. Developing an agile, multi-layer approach to managing resources, where the higher layers set the appropriate policy on a more global (ship) basis, while lower layers monitor and react rapidly to maintain those policies by regulating local subsystem behavior.
2. Developing an adaptive resource management strategy where changes in mission requirements, load, or available resources would lead to rapid reconfiguration with appropriate tradeoffs among the managed properties.

While each of these sub-goals was a significant challenge in its own right, their solution was tightly intertwined for the target shipboard computing environment. The resulting capability became known as dynamic Multi-Layer Resource Management, or MLRM. MLRM was a distinct departure from common practice for Navy shipboard systems, which utilized strictly static resource allocation techniques in a single layer approach. MLRM served as a basis for the Phase 2 efforts of this project described in Sections 3, 0 and 5.

The capability we envisioned could become part of the standard off-the-shelf middleware interposed between the application subsystems and integrated with lower level COTS infrastructure elements. It would serve to provision, configure, monitor and adapt the elements requiring coordinated or controlled actions to achieve the appropriate end-to-end QoS results over a shared resource base. It would automatically select the appropriate real-time property management discipline for the current configuration, regulating other aspects as a side effect. It would scale to the size anticipated for larger versions of PoR class platforms and have a reaction/reconfiguration cycle time commensurate with the requirements from the current complement of sensor and weapons platforms anticipated for the PoR. The middleware resource management algorithms and mechanisms would be parameterized and easily replaceable, to allow additional strategies to be inserted as the ship's configuration and the knowledge of how to construct and run these more dynamic software capabilities evolved over time.

At a more focused granularity, the research goals for Phase 1 were:

- Establishing a multi-layer architecture, supporting dynamic resource decision making, suitable for the PoR computing environment.
- Within that architecture, investigating and instantiating an application-centric management function with dynamic properties for end to end management of application functionality appropriate for changing shipboard conditions.

- Investigating and instantiating a multi-layer resource status service for the shipboard environment which could be used to assess current conditions and drive resource reallocation decisions.
- Providing a standards-based component infrastructure on which to build the dynamic resource management capability to demonstrate the feasibility of using an open COTS base capability for shipboard real-time computing.
- With other ARMS researchers, developing a working prototype for the dynamic multi-layer resource management capability through integration of the various parts of the architecture and integration with the proxy PoR system components.
- Testing, measuring, evaluating and iterating on the proposed solution to ensure its effectiveness for timely dynamic resource management under changing conditions and in processing typical PoR workloads and missions, and to meet the pre-established ARMS Phase 1 Gate Test program performance metrics.

There were three gate tests for the ARMS program in Phase 1: 1) enhanced configuration options, 2) dynamic resource management handling application overload conditions, and 3) dynamic resource management recovering from resource pool failure.

In carrying out these tasks, the BBN team developed and delivered the following MLRM system software components:

1. A resource status service (RSS), customized for the low-layer, mid-layer, and top-layer resource management needed for MLRM.
2. An application string (i.e., collection of integrated applications) manager (ASM) capability and an application proxy capability for providing integrated end-to management of applications.
3. A design time and runtime implementation for the CORBA Component Model (CCM) software engineering paradigm as a COTS base platform for developing the PoR dynamic resource management capability.

As the project progressed, we added an additional task, in conjunction with integrating and evaluating the various components of the MLRM capability. This additional task involved organizing, configuring and supporting the use of the University of Utah Emulab facility as an integration, testing and benchmarking platform for the various ARMS contractors individually and collectively to demonstrate the viability and effectiveness of the ARMS MLRM technology.

2.1 Architecture

2.1.1 Multi-Layer Resource Management

2.1.1.1 Overview

For our Phase 1 effort, the primary decomposition of resources into layers was based primarily on locality. As a design principle, policy directives flow from the upper layers to the lower layers, while the status of the system flows from the lower layers to the upper layers.

The lowest layer is the Resource layer. Each individual resource node is characterized by its particular capabilities and capacities for performing useful work. The entity which performs the work is referred to as an Application. The applications on a particular resource node are controlled via a local Node Provisioner.

The next layer is the Pool layer. Pools collect sets of related resources into a single management domain, overseen by a Pool Manager. Pools are typically defined by locality, both physically and with respect to network topology, but may also take into account resource types, security constraints, etc. The Pool Manager uses a Resource Allocator to determine how best to disperse work across its resource nodes, then works with the Node Provisioners on those nodes to deploy the associated applications.

The next layer is the String layer. Strings are defined as communicating collections of applications that are logically organized to accomplish some set of user-level tasks, or end-to-end capability. Strings will often span pools in order to access specialized resources, balance load, and provide fault tolerance. In some cases applications may be shared across strings, increasing the work done by the application due to increased string communication. Since the Pool layer can only manage the portion of the string within its own pool, the Application String Manager is assigned the responsibility for managing the behavior of the string overall.

The next layer is the Infrastructure layer. This layer is responsible for assigning resources and work to pools. Allocation of work to pools is done in terms of aggregate pool resource availability instead of detailed node level resources, allowing for rapid allocation without the overhead of fine-grained centralized data collection. The management of this layer is performed by the Infrastructure Allocator.

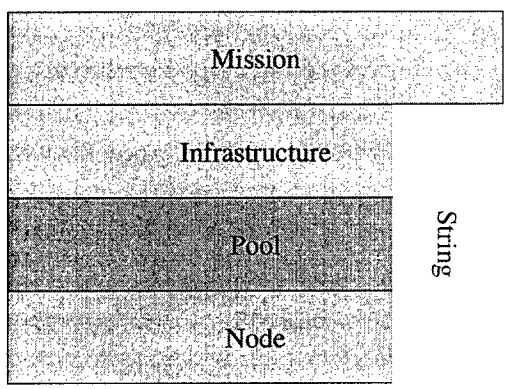

Figure 1: Layers

The top layer is the Mission layer, which defines the work to be done and its relative importance for the current mission. Figure 1 illustrates how these layers conceptually relate to one another, while Figure 2 indicates how the MLRM layers fit within a common multi-layer system software organization.

2.1.1.2 Deployment

A typical mission deployment in this architecture progresses as follows. Some entity in the Mission layer presents the Infrastructure Allocator with an Application String definition. The Infrastructure Allocator examines the requirements of the Applications within the String and the total resources available within each pool and assigns the applications to pools. To maintain consistency across layers, the communicating groups of applications that are assigned to the same pool are bound together as smaller strings, or Substrings. The original String and the Substring/Pool pairs are then passed to the Application String Manager.

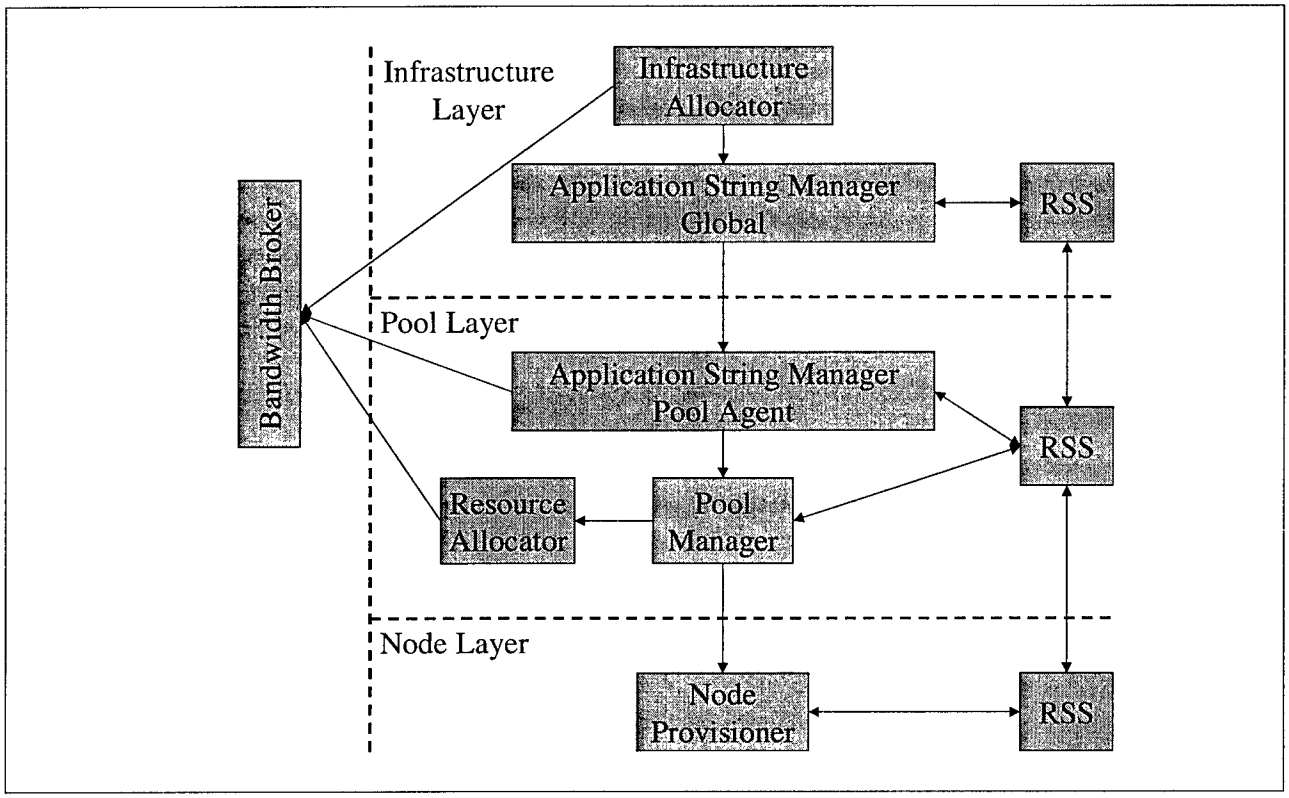

Figure 2: Components

The Application String Manager hands the various substrings to the Pool Manager in their respective pools, which consults a Resource Allocator to select nodes and a number of Node Provisioners to deploy the applications. Application control information is returned to the Application String Manager, which starts the applications in the proper order and returns string control information to the Infrastructure Allocator, which in turn returns string status information to the Mission layer.

2.1.2 Monitoring/Response

To deal with unexpected events during runtime we define three roles:

- Condition Monitor
- Determinator

- Reaction Coordinator

A Condition Monitor gathers any detailed information required to determine that some specific condition has occurred, (e.g., a particular value has exceeded a threshold,) and emits Condition Events. A Determinator takes in one or more Condition Events, determines that a Problem has occurred, and generates Problem Events. (This role includes any future Root Cause Analysis functionality.) A Reaction Coordinator accepts Problem Events and is responsible for driving the system's reaction to the Events.

For both types of Event, we define two classes and four levels. The Event classes are Liveness and Performance, while the levels are Application, Host, String, and Pool. Any given Event has both a class and a level, so e.g., a Condition Event indicating a threshold crossing for CPU consumption by an application is an Application Performance Condition Event. A single Determinator is responsible for generating Problem Events of a particular class at a particular level, and a single Reaction Coordinator is responsible for reacting to Events of that class and level.

To deal with cases where lower level problems should be treated as part of higher level problems, Determinators may be arranged in a hierarchy, with a given Determinator having zero or more parents. Before emitting a Problem Event a Determinator passes the proposed Problem Event to its parent(s), any of which may decide to suppress the problem and optionally use the information in the production of a higher level Problem Event. The suppression logic may be arbitrarily complex, so it may be used to perform a full Root Cause Analysis. However, the amount of delay should generally be kept to a minimum to allow for rapid reaction to lower level problems.

Reaction Coordinators will generally attempt to react to the problem by dealing with services at or below the level of the Problem Event. If the problem cannot be resolved at the given level, (e.g., due to lack of resources) the Reaction Coordinator can take on the role of a Condition Monitor and generate a higher level Condition Event for further processing. A Reaction Coordinator may have several approaches for dealing with a Problem, and some approaches may not always be desirable, so a Reaction Policy is required to determine the applicability and relative desirability of each approach.

2.1.3 Application String Management

The basic notion of Application String Management is that resource management should be focused on maintaining mission capabilities. While the Infrastructure layer has a coarse overall view, and the Pool and Node layers have detailed local views, these layers can't ensure that the needs of the mission are being met when the application string is distributed across pools. In addition, the sort of dynamic tradeoffs we are exploring need to be made with the knowledge of their impact on the affected strings and the mission capabilities underlying them.

This leads to two areas where Application String Management is necessary:

- Monitoring applications for proper behavior relative to the strings in which they participate
- Enforcing string-based policy in MLRM adaptations

As an example of string-based monitoring, consider strings of applications which do work when messages pass between them. If some of the messages are periodic, we can expect that the recipient applications will execute periodically and use a certain amount of CPU resources, which we can monitor and verify. However, if a recipient application is shared across multiple strings, the expected number of messages and CPU resources consumed by the application will increase, so the monitoring needs to be string-aware. Other monitoring, such as end-to-end (or critical path) latency, is inherently string based and reflects directly on the system's ability to meet the requirements of the mission capability.

An MLRM system can encounter a wide variety of undesired behaviors at runtime, including:

- overloaded nodes
- load imbalance across nodes
- overloaded pools
- load imbalance across pools
- node failures
- pool failures

When these things occur, MLRM adapts to deal with them. While it may be possible to resolve such problems by addressing the symptom, it is more appropriate to address their impact on the affected strings. For example, in the case of an overloaded node, a possible initial adaptation is to give priority to the more important applications on the node and deprecate the others. However, the importance of the applications is based on the importance of the strings in which they participate, and deprecating one application in a string is going to impact the rest of the string, so such decisions must take into account string-based policies. For example, it may be the case that a string of lesser importance can't produce useful results in a deprecated mode, in which case it would make more sense to shut it down and free all of its resources for more important strings until sufficient resources are made available.

2.1.3.1 Resource Status Service

Dynamic adaptation to changes in resource availability requires timely, accurate relatively high-level and to some extent domain-specific data. This data generally has to be synthesized from a set of simpler data of variable reliability and timeliness. This is what the Resource Status Service (RSS) provides within the ARMS MLRM architecture.

The core RSS comprises a collection of data definitions, some general and some domain-specific, which form a natural dependency graph rooted at very basic sensor-like data. These definitions and their relationships (the meta-data) are currently specified in advance, at compile time, and are a natural candidate for code-generation from a more abstract data model, though this kind of linkage hasn't yet been made. The meta-data could be made accessible at runtime as well, for instance to define new relationships or refine old ones on the fly, if it seems useful to do so.

The current runtime interfaces to the RSS correspond directly to inputs and outputs represented by the data graph. Inputs supply values for root nodes of the graph and take the form of simple tagged data. These can arrive and be processed at a very high frequency. Outputs, either in the form of an in-band response to a query or an out-of-band callback to a subscriber, provide high-level data values to the ultimate consumers. The transformation chains between lower and higher level data values can either happen on-demand or in the background.

The efficiency requirements of the RSS make it more useful as a local service (i.e., as part of a running application or as a standalone application per host) than as a single global, shared service. At the same time, experiments have shown that a collection of RSS instances in a distributed system can share data with one another without excess overhead by using the idea of "gossip" – extra data piggybacked on the messages that are already flowing between the distributed system's components anyway. The result of using gossip in this way is a highly extensible distributed resource status service at a comparatively low cost.

2.2 Implementation Development Activities

2.2.1 Overview

The implementation of the MLRM is oriented towards the CORBA Component Model. The various MLRM components are linked together using facets and receptacles for the primary operations, with event ports used to communicate system status. The components are deployed in hierarchical groups based on locality. A "Global" (or Coordinator) group includes the Infrastructure Allocator, the Security Provisioner, and the Application String Manager Global components. A "per-Pool" group includes the Application String Manager Pool Agent, Pool Manager, and Resource Allocator components. A "per-Node" group includes the Node Provisioner component. The Global components are placed in one assembly and the per-Pool and per-Node components in another, with additional tools connecting components from the two assemblies.

Because they are based on existing tools, the Resource Status Service and Bandwidth Broker components don't follow the CORBA Component Model, instead using a CORBA Naming Service to publish their services.

The source code for all of the components, shared libraries, and middleware was stored in a common CVS repository to track revisions and facilitate integration.

Development was directed towards satisfying the following ARMS Phase 1 Gate Tests:

- GM1 – Multiple Configurations
- GM2 – Increased Capability
- GM3 – Fault Tolerance

GM1 was met by demonstrating basic dynamic deployment capabilities. GM2 was met by detecting the increased resource demands imposed by increasing capability and adjusting less important activities to compensate in order to stabilize the system. GM3 was met by automatically redeploying application strings in a scenario in which static failover mechanisms were unable to operate.

2.2.2 Application String Manager (ASM) Functionality

2.2.2.1 ASM Capabilities

The Application String Manager software had the following capabilities (and the primary Gate Tests which required them) as of the end of Phase 1:

- Deploy Applications Strings (GM1)
- Start Applications within an Application String in Specified Startup Order (GM1)
- Deploy Application Strings with Substrings (GM2)
- Deploy Application Strings with Substrings to Multiple Pools (GM2)
- Configure Network Bandwidth between Pools (GM2)
- Deploy Substrings within Pools (GM2)
- Configure Condition Monitors on Shared Applications (GM2)
- React to Application Overloads by Deprecating Competing Strings of Lesser Importance (GM2)
- Deploy Application Strings with Replica Applications (GM3)
- Configure Condition Monitors on Pools (GM3)
- Detect Pool Liveness Problems (GM3)
- Redeploy Application Strings to deal with Pool Failures (GM3)

2.2.2.2 ASM Interactions

The Application String Manager interacts with the following MLRM components:

Component	*Gets from ASM*	*Sends to ASM*
Infrastructure Allocator	String Deployment Status, Pool Failure Problem Events	String Deployment and Redeployment Directives
Pool Manager	Substring Deployment and Reconfiguration Directives	Substring Deployment Status Events, Application Proxy References

Component	*Gets from ASM*	*Sends to ASM*
Bandwidth Broker	Inter-Pool Network Reservation Requests	Network QoS Settings
Node Provisioner	Application Start, Monitor, and Network QoS Directives	
Resource Status Service	Subscription Requests	Application Load, Host Liveness, and Pool Liveness Data
Security Provisioner	MLRM Component Registration	

2.2.2.3 ASM Implementation

The Application String Manager (ASM) is implemented in C++ as two CIAO components:

- ASM Global (ASM-G)
- ASM Pool Agent (ASM-PA)

The former is responsible for managing entire strings, while the latter is responsible for managing the portion that resides within a single pool.

The ASM components act as normal CIAO components with a few caveats:

1. Anticipating that pools will need to come and go at run time, and given that the current CIAO tools didn't allow for dynamic changes to assemblies at the time, we don't use the standard assembly tools to connect ASM-G to the various ASM-PAs. Instead they are connected using the `ASM_connect` utility, which takes object references for the ASM-G and an ASM-PA with its associated pool-id and makes the appropriate calls to connect the two. (The current state of CIAO would allow us to use more standard CCM interfaces (i.e., multiplex receptacles); we just haven't made the change yet.)
2. Object references for some non-CCM CORBA services, namely the Bandwidth Broker and Resource Status Service are accessed through a name service instead of through a receptacle.
3. ASM relies on a third party to initialize the log4cplus library before we are started, so a LoggingInit component needs to be included in any `<processcollocation>` which contains an ASM component.

The source code for the ASM interfaces and implementation reside in the ARMS CVS repository under:

```
DRM/DRM_Services/ApplicationStringManager/Simple-BBN
```

The IDL which defines the ASM interfaces resides in:

- `ApplicationStringManagerGlobal.idl`

- `ApplicationStringManagerPoolAgent.idl`
- `ApplicationStringInstanceManagement.idl`
- `ApplicationSubstringInstanceManagement.idl`
- `StringOverloadDeterminator.idl`

The first two define the component interfaces. The second two define the facet interfaces used by the components. The last one is not used; it is provided only as a possible future component interface for a `StringOverloadDeterminator` component separate from ASM-G.

2.2.3 Resource Status Service (RSS) Functionality

2.2.3.1 RSS Capabilities

The Resource Status Service provides a common service for acquiring, providing and aggregating status information pertinent to monitoring and controlling applications and the system itself, including individual components. It serves as the common sensor component for MLRM, by providing the common framework for acquiring status data from particular resources (including hardware resources, system (software) resources, and application level resources), and for providing various clients with customizable and periodic "reports" updating current status. Included in the status information is "heartbeat" collection from the active software elements of the system (and applications) as a proxy or indication that a component is still running and operating correctly. Clients subscribe to various status data feeds from the common RSS to get periodic updates based on various events, and use this information to make evaluation and reconfiguration decisions. The frequency of collecting and reporting status is configurable. In addition, capabilities are provided for aggregating collections of data (often from lower level resources) and providing a higher level integrated or summary view across a variety of resources types. The RSS service tries to efficiently serve many clients, often accessing common subsets of data, and at the same time try to optimize the method and form of delivery to satisfy real-time delivery requirements. To do this, the RSS uses a distributed implementation, with elements of the RSS cooperating with each other to provide common access to dispersed data, and to expedite collection and delivery of remote data to dispersed clients. The RSS has also been used as a shared transaction status repository for resource management allocations performed in part by multiple resource managers.

2.2.3.2 RSS Interactions

The RSS itself does not supply any data. It depends on other tools, software sensors, to do that. These sensors have to exist and be running wherever needed, e.g., on hosts if they're gathering host resource data (CPU usage, load average etc) or on routers if they're gathering network resource data (bandwidth etc.). Existing sensor tools can be linked into the RSS via CORBA. But for some domains, new sensors will have to be written and deployed.

2.2.3.3 RSS Implementation

The RSS is currently implemented as CORBA-accessible Java code.

2.3 Laboratory Support

The target integration, testing and experimentation environment for ARMS Phase 1 was the Operational Experimentation Platform provider's (OEP, which was the Raytheon Company) ARMS Integration Facility (AIF). However, the AIF had a number of problems which limited its usefulness:

- Initially there was no remote access, and remote access remained limited and difficult for some non-OEP personnel.
- Each machine was configured differently, making it difficult to run tests.
- The machines were also used for development, which conflicted with testing.
- There was only a single set of machines, limiting the lab to a single experiment or test at a time and creating scheduling issues.

To provide a stable environment for initial integration and testing for all Technology Developers (TDs), BBN took on the responsibility of setting up and maintaining an ARMS project within the University of Utah's Emulab[1] system for dynamically allocating and configuring collections of integrated nodes into a testbed. This involved:

- Configuring host node operating system images
- Configuring network node operating system images
- Setting up builds of the ARMS middleware
- Setting up builds of the OEP and MLRM software
- Setting up MLRM disk images
- Producing a prototype network configuration (experiment)

[1] http://www.emulab.net/

The use of the Emulab was a great success, with the majority of the TDs using it at one point or another, and a number using it on a regular basis. Without the initial integration and testing in the Emulab the integration, testing, and experimentation in the AIF would have gone much less smoothly, and the program would have been at a much greater risk.

The only significant issue we faced with the Emulab was that of resource contention. Since the Emulab is shared among a number of different projects, during the busier periods there were occasionally an insufficient number of resources to run ARMS experiments. This could be easily remedied by funding additional Emulab resources or by setting up a separate Emulab. [This was in fact done in ARMS Phase 2.]

2.4 Integration and Testing

Given that the Application String Manager is at the center of the collection of MLRM components, it was natural for BBN to do much of the initial integration. The integration generally involved the following steps:

- Building the middleware
- Building the OEP software and addressing build problems
- Building the MLRM software and addressing build problems
- Creating CIAO assembly descriptor files for the MLRM components
- Creating startup scripts to deploy the middleware, OEP, and MLRM components
- Running the startup scripts and addressing startup problems
- Deploying application strings using MLRM and addressing execution problems
- Running additional tests of advanced features

This integration was generally performed in the Emulab since the environment was already set up and generally available.

BBN was also heavily involved in the final integration and testing in the AIF. For each Gate Test we made at least one multi-day trip to Portsmouth, RI, to work with other TDs and the OEP to get the software to where it could run experiments to demonstrate Gate Test functionality. We also provided regular support to efforts in the AIF remotely.

2.5 Conclusion

Phase 1 of the ARMS program was focused on deriving a common architecture for dynamic multi-layer resource management consistent with the configurations and applications emerging from the next generation Navy surface ship domain, and having the various TDs, with various spheres of expertise, contribute components to instantiate that architecture and design sufficient to have it undergo test and evaluation against program metrics. The BBN team's specific focus beside initiating many of the driving architectural concepts was in providing application and end-to-end application string management components, developing a system service for collecting and disseminating resource status information system wide, providing a standards based software platform for easily linking and connecting the various MLRM software components, leading system integration, test and evaluation activities, and introducing and supporting a commonly accessible testbed facility to significantly improve multi-TD integration and testing activities.

Together with other program participants, we were successful in developing a prototype MLRM subsystem sufficient to demonstrate and measure dynamic resource management operating against simulated PoR workloads, and effectively managing a wide variety of alternative configurations, managing application overload to maximize resources applied to high priority tasks, and recovering from large scale failure. These prototype capabilities were evaluated against pre-established program metrics. They showed sufficient maturity and continued potential to provide the intended risk reduction attributes for developing similar operational surface ship capabilities, to warrant an ARMS Phase 2, which is described in the following sections.

3. Fault Tolerance Research, Experimentation, and Evaluation

3.1 Introduction to ARMS Fault Tolerance Activities and Results

Fault tolerance (FT) is a crucial design consideration for mission-critical distributed real-time and embedded (DRE) systems, such as the MLRM. These systems combine the real-time characteristics of embedded platforms with the dynamic characteristics of distributed platforms. However, many of the characteristics of these systems, such as heterogeneity, strict timing requirements, scalability, and non client-server application interactions, prove challenging to implementing a fault-tolerance solution with the techniques and technology that existed at the beginning of ARMS. In order to make the MLRM fault tolerant, we had to design and implement several innovative advancements to the state of the art in fault tolerance. These advancements included enabling the cooperative use of group and non-group communications, improving efficiency and scalability; developing fault tolerance support for CORBA components and their peer-to-peer calling patterns; developing dynamic deployment of CORBA components; supporting multiple languages (C++ and Java); and supporting multiple replication schemes; among other advances. Section 3.2 describes our success in meeting these challenges and fulfilling the ARMS Gate Test 3 requirements, which established an ARMS program wide goal of creating a high performance fault tolerant MLRM that exceeded the PoR RM recovery requirements. Section 3.3 describes the R&D that enabled this success in detail.

3.2 Gate Test 3 – Fault Tolerance of Dynamic Resource Manager

In Phase I of ARMS, we created a prototype implementation of a Multi-Layer Resource Management (MLRM) framework. In Phase II we made the MLRM framework fault-tolerant through a combination of research and implementation of dynamic, real-time, fault tolerance. We were guided in this undertaking by the ARMS Gate Test 3 requirements and ran a number of experiments and evaluations, described in this section. The experiments and evaluations enabled us to quantify our claims that the MLRM is fault-tolerant under a number of scenarios, some above and beyond the capabilities of the PoR resource manager requirements.

3.2.1 Introduction and Summary of Results

Gate Test 3 was officially defined on June 1, 2005, as one of four ARMS Gate Tests in the *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*. The purpose of this Gate Test was to show that we could make the MLRM system developed in ARMS Phase I fault tolerant (in a manner similar to the requirements of the *PoR*), meet the PoR's recovery time requirement, and handle faults (cascading failures) beyond those currently required by the PoR.

In this section, we describe the final results of the Gate Test 3 experiments, supplying values for the metrics in the Gate Test document, as well as several results that go above and beyond the descriptions in the Phase II Experimentation Plan document. Incremental steps and intermediate results can be found in earlier reports on the GateTest3 Wiki page, https://repo.isis.vanderbilt.edu/twiki/bin/view/ARMS/GateTest3.

Gate Test 3 was divided into a Scenario 3A, whose purpose was to show that we could meet or exceed the recovery scope and time requirements of the PoR, and a Scenario 3B, whose purpose was to show that we could exceed the failure condition recovery requirements of the PoR.

3.2.1.1 Summary of Gate Test 3 Results

This section shows that we

- *Satisfy* the letter of the law Gate Test 3A requirements;
- *Significantly exceed* the Gate Test 3A requirements using ARMS fault tolerant research technology applied to MLRM functionality comparable to the PoR Ensemble Infrastructure Resource Manager (mIRM) functionality;
- *Satisfy* Gate Test 3B requirements to survive cascading failures not required by the PoR; and
- *Exceed* the letter of the law of Gate Test 3 with extra capabilities including recovering the fault tolerance level after failure, recovery using hardware similar to the PoR (ISISlab where we obtained 16x faster recovery than the PoR's recovery requirement), and significantly exceeding the Gate Test recovery requirements by optimizing our open-source commercial database elements.

3.2.1.2 Organization of This Section

In Section 3.2.2, we start by revisiting the Gate Test 3 official definition, which was defined as an *Overview* and as an *Elaborated Scenario* for Gate Tests 3A and 3B, and provide a point-by-point description of how the Gate Test was conducted to meet the requirements of the definition. This section also provides the metrics that the Gate Test 3 definition specifies must be collected and the results we collected. This section is purposefully written to address point-by-point the *letter of the law* definition of the Gate Test and retains the redundancy in the originally definition document. The casual reader might find this section repetitive and might want to read only the *Overview* sections, Sections 3.2.2.1 and 3.2.2.2, and then skip ahead to Section 3.2.3.

Section 3.2.3 provides a more detailed set of results and analysis of these results. Section 3.2.4 provides a description and results of the extras we did for Gate Test 3, i.e., things we did above and beyond the letter of the law of the Gate Test definition. In this section, we describe three of them: replica reconstitution to get back to an acceptable level of fault tolerance after a failure; running the experiments on the ISISlab, which is more representative of the PoR's environment; and tuning the Bandwidth Broker database recovery to get higher performance from the commercial database used by the Bandwidth Broker.

Section 3.2.5 describes the results of rerunning the Gate Test 3A and 3B experiments on the ISISlab, with the tuned commercial database, and analyzes the results.

Finally in Section 3.2.6, we draw some conclusions for the Gate Test 3 experiments.

3.2.2 Definition of Gate Test 3 and Point by Point Results

The official definition of Gate Test 3 from the *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan* includes both an **Overview** and an **Elaborated Scenario** each for Gate Test 3A and 3B. In the following sections, we repeat each of these *verbatim* from that document and provide a point-by-point description in *italics* of how we executed and passed the Gate Test.

Gate Test 3A was defined to show that we could provide fault tolerance for the ARMS MLRM that *meets* the PoR's requirements for fault tolerance of their mIRM. Specifically it states that we could make the infrastructure elements of MLRM recover from a single pool failure within the time requirement defined by the PoR. The metrics for Gate Test 3A ask (1) can MLRM recover from a pool failure and (2) how fast compared to the PoR's mIRM recovery requirement?

Gate Test 3B was defined to exhibit that we could make the ARMS MLRM recover from faults beyond those required by the PoR, specifically that we could survive the failure of two MLRM instances (the operational one and a partially recovered replacement) in rapid succession. The metrics for Gate Test 3B ask (1) can MLRM recover from two cascading pool failures and (2) within what time (for information only, since it has no comparable baseline requirement).

3.2.2.1 Gate Test 3A Overview and Point by Point Results

Test Scenario 3A (MLRM meets Program needs) - Overview (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

> We will replicate MLRM across multiple data centers in a fashion similar to that for mIRM (the ensemble Infrastructure Resource Manager in the baseline PoR design) in the Release 3 System Acceptance Test.
>
> We will fail one of the data centers (in the fashion of one of the major Release 3 System Acceptance Test failure scenarios).
>
> Observe whether the MLRM recovers within the time required of the mIRM.

Test Scenario 3A Overview Point by Point Results

1. *To complete this gate test we replicated the global MLRM management components: the IA/ASM-Global (IA/ASM-G), the bandwidth broker (BB), and the RSS. The IA/ASM-Global was actively replicated while the BB and RSS were passively replicated. The BB, in addition to ARMS FT technology, makes use of an open-source commercial database (MySQL). We engineered its cluster feature to satisfy our recovery semantics. Specifically, any cluster partition could take over after a failure.*

2. *In order to simulate a catastrophic pool failure, as when battle damage destroys the whole pool, we stopped network traffic at the router passing traffic to and from the pool. This instantaneously cut off the pools from one another without giving the OS or network stack any opportunity to communicate failure to the other side of any network connections.*

3. *The MLRM recovered, which we showed by successfully redeploying the application strings.*

Metrics: (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. Does the MLRM recover its functionality? (Boolean)

2. Does recovery time of MLRM meet or exceed the Flight 1 recovery time requirement for the TSCE-I's Resource Manager for the Release 3 System Acceptance Test?

Test Scenario 3A Overview Metrics Point by Point Results

1. *Yes (True).*

2. *Yes. ARMS MLRM management functionality recovered in an average of 60 ms (worst case 90 ms) and Bandwidth Broker Database functionality recovered in an average of 212 ms (worst case 283 ms). Both numbers are under the PoR recovery requirement time. In the case of the elements using ARMS technology (the management functionality), recovery is significantly under the PoR recovery requirement time. Specific numbers are shown below in Section 3.2.3.2.*

3.2.2.2 Gate Test 3B Overview and Point by Point Results

Test Scenario 3B (MLRM provides additional capabilities) - Overview (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. We will replicate MLRM across multiple data centers in a fashion similar to that for mIRM (the ensemble Infrastructure Resource Manager) in the Release 3 System Acceptance Test.

2. We will fail one of the data centers followed by an additional failure in an MLRM component in a surviving data center.

3. Observe whether the MLRM recovers.

Test Scenario 3B Overview Point by Point Results

1. *The MLRM was replicated in the same manner as in 3A, with one additional set of replicas on the third pool.*

2. *The pool failures were carried out in the same manner as 3A. In order to simulate a cascading failure, rather than just two failures one right after another or two failures at the same time, we placed a delay between shutting down the two pools and made sure the failures were cascading by post-processing the results and throwing out any runs which did not contain a cascading failure.*

3. *The MLRM recovered, which we showed by successfully redeploying the application strings.*

Metrics: (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. Does the MLRM recover its functionality? (Boolean)

2. Time of recovery of MLRM functionality.

Test Scenario 3B Overview Metrics Point by Point Results

1. *Yes (True).*

2. *ARMS MLRM management functionality recovered in an average of 47 ms (worst case 51 ms) and Bandwidth Broker Database functionality recovered in an average of 509 ms (worst case 580 ms). Specific numbers are shown in Section 3.2.3.3.*

More detailed analysis of the Gate Test 3A and 3B results is provided in Sections

3.2.2.3 and 3.2.2.4. A comparison of the results of 3A and 3B is in Section 3.2.3.4.

3.2.2.3 Elaborated Scenario 3A and Point by Point Results

Elaborated Scenario 3A (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. We will replicate MLRM across multiple data centers in a fashion similar to that for mIRM (the ensemble Infrastructure Resource Manager for the target program) in the Release 3 System Acceptance Test (The target program mIRM uses a Master-Slave replication strategy).

   Two pools of 3 processors each representing 2 data centers will be operational.

   Deploy the master of Infrastructure Allocator (IA), Application String Manager (ASM) Global, and Bandwidth Broker on a single machine, with replicas (slaves) on a different machine in a different pool.

   10 application strings representing a mixture of mission critical (7) and mission support (3) functions have been deployed (this represents a realistic MLRM state)

2. We will fail one of the data centers (in the fashion of one of the major Release 3 System Acceptance Test failure scenarios).

   1. The resource pool containing the master MLRM elements is failed catastrophically

2. The Pool Failure Condition Monitor will detect the pool failure and generate a pool failure event

3. Observe whether the MLRM recovers within the time required of the mIRM.

    1. Pool Failure Response Coordinator receives the pool failure event and directs slave instances of IA, ASM, and Bandwidth Broker to become masters

    2. IA, ASM, and Bandwidth Broker slave elements become promoted to master elements

Elaborated Scenario 3A Point by Point Results

1. *We replicated the MLRM using active replication for the IA/ASM-G and passive for the RSS and BB.*

    1. *We used two pools with three hosts in each pool*

    2. *We deployed replicas of the MLRM components in each pool. The primary passive replicas were placed on the failed pool so that there would be a fail-over event when a failure occurred. As there is no primary/backup distinction among active replicas there was no need to start them in a particular order.*

    3. *We deployed 10 strings as described above.*

2. *As described in Section 3.2.2.1, we failed the pool by turning off routing for the pool.*

    1. *The routing failure is a catastrophic failure; the whole pool dies in an instant.*

    2. *The MLRM did report a pool failure event.*

3. *The time of recovery is reported above in Section 3.2.2.1 and in depth in Section 3.2.3.*

    1. *Our replication middleware noted the failure and made the backup passive replicas primaries. The actively replicated IA/ASM-G noted that the lost pool's replica was gone.*

Data to Be Measured / Logged (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

Recovery start time: When the Pool Failure Response Coordinator receives the pool failure event

Recovery end time: When all of the IA, ASM Global, and Bandwidth Broker slave elements have become masters

Elaborated Scenario 3A Data to Be Measured / Logged Point by Point Results

1. *To measure the worst-case recovery times, we noted all the times that MLRM elements in the remaining pool noticed the failure and logged the earliest value as the recovery start time (receipt of the pool failure event).*

2. *For both active and passive components we log the recovery end time as the time they are ready to process new messages. For active replicas, this is the time at which the group communication system (GCS) is able to process messages from MLRM elements. For passive replicas, we logged the time at which the GCS is able to process messages plus the time it took to promote a backup to primary. For the BB DB we measured the time from detection until a query was successfully completed.*

Approach to Compute Test Metrics (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. Metric 1 (Boolean)

    1. True if all of IA, ASM Global, and Bandwidth Broker have master elements at the end of the experiment, False otherwise

2. Metric 2 (Real) Time of recovery

    1. Computed by subtracting the Recovery start time from the Recovery end time

    2. Compare to Flight 1 recovery time requirement for the TSCE-I's Resource Manager for the Release 3 System Acceptance Test

Elaborated Scenario 3A Test Metrics Point by Point Results

1. *True.*

2. *ARMS MLRM management functionality recovered in an average of 60 ms (worst case 90 ms) and Bandwidth Broker Database functionality recovered in an average of 212 ms (worst case 283 ms). Both numbers are under the PoR recovery requirement time. In the case of the elements using ARMS technology (the management functionality), recovery is significantly under the PoR recovery requirement time. Specific numbers are shown in Section 3.2.3.2.*

Envisioned Test-bed Environment (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. 6 nodes, 2 pools, in Emulab environment with

    1. 10 application strings, 15 apps per application string

Elaborated Scenario 3A Test-bed Environment Point by Point Results

1. This was done exactly as listed.

    1. We used 10 application strings for 15 apps each as noted above.

3.2.2.4 Elaborated Scenario 3B and Point by Point Results

Elaborated Scenario 3B (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. We will replicate MLRM across multiple data centers in a fashion similar to that for mIRM (the ensemble Infrastructure Resource Manager) in the Release 3 System Acceptance Test.

    1. 3 pools of 3 processors each representing 3 data centers

    2. Deploy the master of IA, ASM Global, and Bandwidth Broker on a single machine, with replicas (slaves) on different machines in each pool

    3. 10 application strings representing a mixture of mission critical (7) and mission support (3) functions have been deployed (this represents a realistic MLRM state)

2. We will fail one of the data centers followed by a second failure.

    1. The resource pool containing the master MLRM elements is failed catastrophically

    2. The Pool Failure Condition Monitor will detect the pool failure and generate a pool failure event

    3. While the first set of slaves is in the process of recovering to master, we introduce an additional failure in the recovering MLRM slaves

    4. The Recovery Failure Condition Monitor will detect the second failure and generate a failure event

3. Observe whether the MLRM recovers.

    1. Recovery Failure Response Coordinator receives the failure event and directs tertiary slave instances of IA, ASM, and Bandwidth Broker to become masters

2. IA, ASM, and Bandwidth Broker slave elements become promoted to master elements

Elaborated Scenario 3B Point by Point Results

1. *We replicated MLRM using active replication for the IA/ASM-G and passive replication for the RSS and BB.*

    1. *We used three pools with three hosts in each pool as noted.*

    2. *We deployed replicas of the MLRM components in each pool. The primary passive replicas were placed on the first pool to fail so that there would be a fail-over event when a failure occurred. As there is no primary/backup distinction among active replicas there was no need to start them in a particular order.*

    3. *We deployed 10 strings as described above.*

2. *As described above we failed the pool by turning off routing for one pool followed by failing the next pool after a small wait (130ms). We looked at the logs and if the failures were reported as two independent failures we threw those logs away. We saved values that had a single failure of two pools knowing that in this case it was a cascading failure due to the wait between failures. The wait value was experimentally determined to give a good chance of having the second failure occur just as the first failure was about to be reported and detected.*

    1. *The routing failures are catastrophic failures.*

    2. *The MLRM did report a pool failure event.*

3. *Timing values can be found in Section 3.2.3.3.*

    1. *Our replication middleware noted the failure and made the backup passive replicas primaries. The actively replicated IA/ASM-G noted that the lost pools' replicas were gone.*

Data to Be Measured / Logged (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

1. Recovery start time: When the Pool Failure Response Coordinator receives the pool failure event

2. Recovery end time: When all of the IA, ASM Global, and Bandwidth Broker slave elements have become masters

Elaborated Scenario 3B Data to Be Measured / Logged Point by Point Results

1. *To show the worst-case recovery times, we noted all the times that MLRM elements in the remaining pool noticed the second failure and logged the earliest value as the recovery start time (receipt of the pool failure event).*

2. *For both active and passive components we log the recovery end time as the time they are ready to process new messages. For active replicas, this is the time at which the group communication system (GCS) is able to process messages from MLRM elements. For passive replicas, we logged the time at which the GCS is able to process messages plus the time it took to promote a backup to primary. For the BB DB we measured the time from detection until a query was successfully completed.*

Approach to Compute Test Metrics (from Adaptive and Reflective Middleware Systems Phase II Experimentation Plan, 1 June 2005)

1. Metric 1 (Boolean)
    1. True if all of IA, ASM Global, and Bandwidth Broker have master elements at the end of the experiment, False otherwise
2. Metric 2 (Real) Time of recovery
    1. Computed by subtracting the Recovery start time from the Recovery end time

Elaborated Scenario 3B Test Metrics Point by Point Results

1. *True.*

2. *ARMS MLRM management functionality recovered in an average of 47 ms (worst case 51 ms) and Bandwidth Broker Database functionality recovered in an average of 509 ms (worst case 580 ms). Specific numbers are shown below.*

Envisioned Test-bed Environment (from *Adaptive and Reflective Middleware Systems Phase II Experimentation Plan*, 1 June 2005)

- 9 nodes, 3 pools, in Emulab environment with 10 application strings, 15 apps per application string

Elaborated Scenario 3B Test-bed Environment Point by Point Results

- *This was done exactly as listed. We used 10 app strings for 15 apps each as noted above.*

More detailed analysis of the Gate Test 3A and 3B results is provided in Section 3.2.3. A comparison of the results of 3A and 3B is in Section 3.2.3.4.

3.2.3 Detailed Gate Test 3 Results and Analysis

This section presents more detailed results for Gate Test 3 and analysis of the results.

3.2.3.1 Explanation of the Two Different Recovery Measurements

In the results presented above in Section 3.2.2 and in this Section, we present the results separately for:

- The MLRM Management elements which includes the top-level MLRM elements, namely the Infrastructure Allocator (IA), Application String Manager-Global (ASM-G), Bandwidth Broker (BB), and the Resource Status Service (RSS)
- The MLRM Management elements plus the Bandwidth Broker Database (BB DB), an open-source commercial database (MySQL).

We treat the BB DB recovery time separately from the management recovery time for the following reasons:

1. The MLRM is operational and able to deploy application strings without the BB present, which means that MLRM critical functionality can be considered recovered with or without the BB DB recovered.

2. The PoR mIRM does not have a COTS DB component. The "Management" numbers provide a better apples-to-apples comparison to the recovery requirement time.

3. The BB DB does not employ ARMS FT technology for its fault tolerance, instead employing MySQL's fault tolerance features, which were not designed for real-time behavior. In the Icing section, we show the results of additional efforts to tune the BB DB recovery time, which resulting in vastly improved failover times.

The first measurement (MLRM Management recovery time) illustrates better the results of ARMS Gate Test 3 fault tolerance research and development. The second number also includes an element of engineering an open-source commercial product.

3.2.3.2 Detailed Gate Test 3A Results and Analysis

Results from our 3A runs on emulab can be seen in Table 1 and Figure 3. MLRM management elements, replicated using ARMS active and passive fault tolerance technology, recovered their functionality on average within 60.42 ms and in worst case in 89.50 ms. This is well below (less than one third) the recovery requirement that we were targeting.

Including the Bandwidth Broker database, a commercial database (MySQL) replicated using MySQL's clustering features (with changes to satisfy our recovery semantics), MLRM with the BB DB recovered on average in 212.48 ms and in worst case within 283.20 ms. This is also below the PoR's recovery requirement.

Table 1: Results from Gate Test 3A runs on Emulab

	MLRM Management	MLRM including BB DB
Average recovery time (ms)	60.42	212.48
Minimum recovery time (ms)	43.90	150.10
Maximum recovery time (ms)	89.50	283.20
Standard Deviation (ms)	20.02	52.39

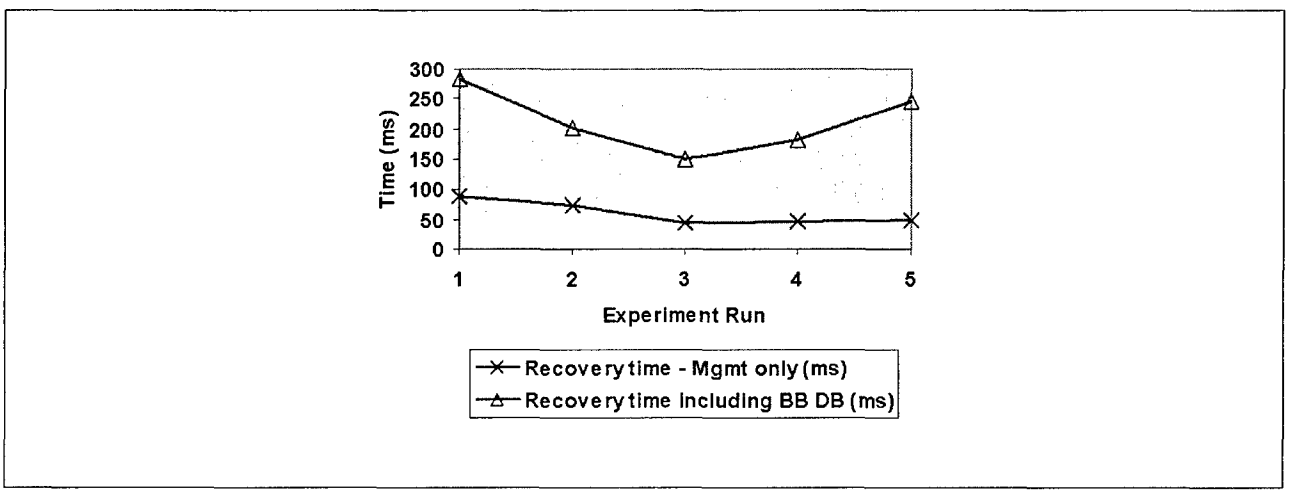

Figure 3: Recovery Time for Five GT3-A Runs on Emulab

Note that we make use of MySQL clustering rather than MySQL replication. This is due to the fact that in MySQL replication, consistency between replicated databases is not guaranteed while the clustering service does guarantee consistency. For example, when using MySQL replication, a transaction may be complete on a master replica and if that replica fails the newly elected master may not know of the transaction. These kinds of problems are avoided using the clustering solution.

The results of Gate Test 3A show that ARMS MLRM Management functionality, made fault tolerant using ARMS fault tolerance research, significantly exceed the PoR recovery requirement. The BB DB functionality, made fault tolerant by ARMS engineering using commercial technology, meet the PoR recovery requirement time.

Therefore, we met the Gate Test 3A requirements with the full MLRM system (including the BB DB) and *greatly exceeded them with the ARMS Fault Tolerant research technology.*

3.2.3.3 Detailed Gate Test 3B Results and Analysis

The Gate Test 3B experiment evaluates whether MLRM can recover from two cascading pool failures. In the experiments, we injected the first fault in the same place and manner as Gate Test 3A, and injected the second fault in as close to the worst case time as we could, i.e., after the system is far along in its recovery, but just before it is completely recovered so that the faults cannot be handled as two distinct faults. Gate Test 3B was defined to be passed if we could answer "Yes" to the first metric, i.e., that MLRM could survive two cascading failures.

In all experiments, MLRM was able to recover from both failures.

The second metric for Gate Test 3B is simply a measure of how fast we could recover MLRM functionality. Results from our experimentation on Emulab can be seen in Table 2 and Figure 4.

ARMS MLRM management functionality recovered from a two-level cascading failure in an average of 47 ms (worst case 51 ms) and the Bandwidth Broker Database functionality recovered in an average of 509 ms (worst case 580 ms).

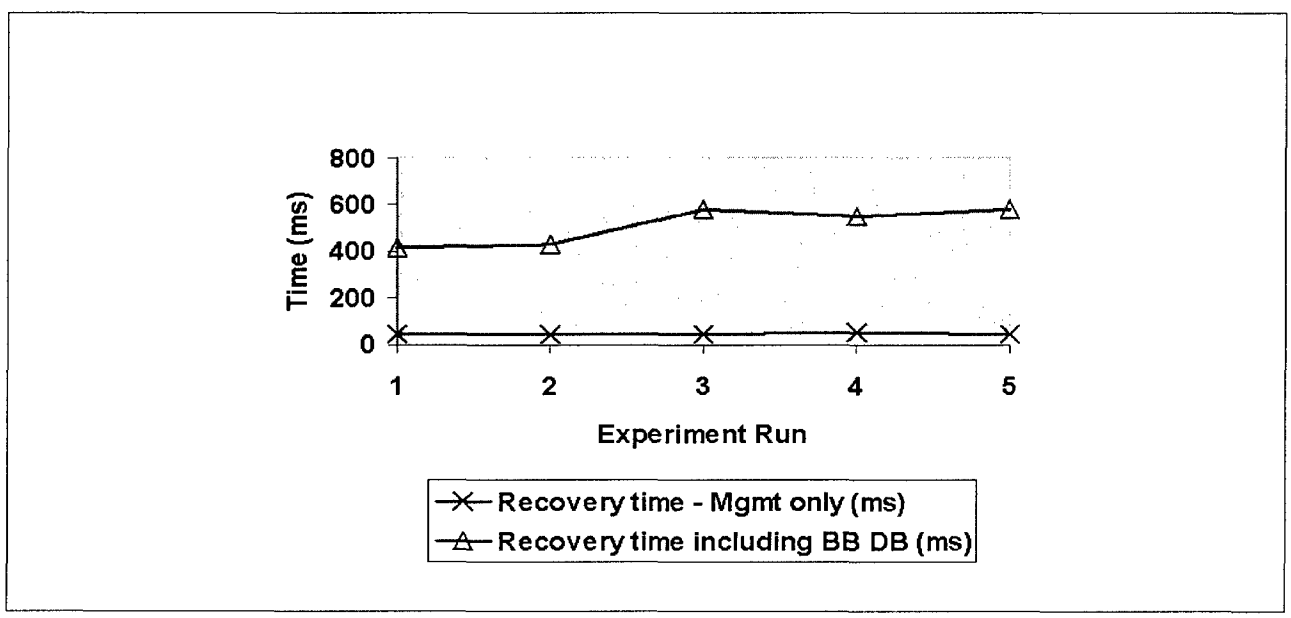

Figure 4: Recovery Time for Five GT3-B Runs on Emulab

The results of Gate Test 3B show that ARMS MLRM Management functionality, made fault tolerant using ARMS fault tolerance research, can survive multiple, cascading failures.

Therefore, we have met the Gate Test 3B requirements.

Table 2: Results from Gate Test 3B runs on Emulab

	MLRM Management	MLRM including BB DB
Average recovery time (ms)	46.96	509.06
Minimum recovery time (ms)	43.00	414.90
Maximum recovery time (ms)	51.00	579.60
Standard Deviation (ms)	3.02	50.85

3.2.3.4 Comparison of Gate Test 3A and 3B Recovery Measurements

From the numbers above, it appears that the recovery time for MLRM management functionality decreases from Gate Test 3A to Gate Test 3B, from 60.4ms to 47.0ms. This is a measurement artifact due to the way we measure recovery. In both sets of Gate Test experiments, we separated the "recovery" time from the "detection" time in the following manner:

- The end of "detection" time was the *smallest* of the time to detect by any of the MLRM elements, i.e., the earliest that any replica detected that a fault had occurred.
- The end of "recovery" time was the *largest* of any of the times to recover of the MLRM elements, i.e., the latest that any element had a replica ready to perform its functionality.

Table 3: Gate Test 3 Recovery Time

Experimental Run	3A IA/ASM (ms)	3B IA/ASM (ms)	3A BB (ms)	3B BB (ms)	3A RSS (ms)	3B RSS (ms)
1	0	0	0.5	0.5	39.7	47.9
2	0	0	0.5	1.0	39.5	45.2
3	0	0	0.6	0.5	43.0	44.3
4	0	0	0.6	0.5	44.8	43.9
5	0	0	0.5	0.5	47.4	42.2

This means that the "recovery" time we are reporting is the absolute *worst case*, the difference between the earliest detection and the latest recovery, from among all MLRM elements. As an example, if the IA/ASM-G process detected the failure first, that's the reported MLRM detection time, even if other MLRM elements haven't detected the fault yet. If the RSS process is the last to recover, the time it is ready to run is the reported time at which MLRM has recovered, even if the IA/ASM-G recovered well before.

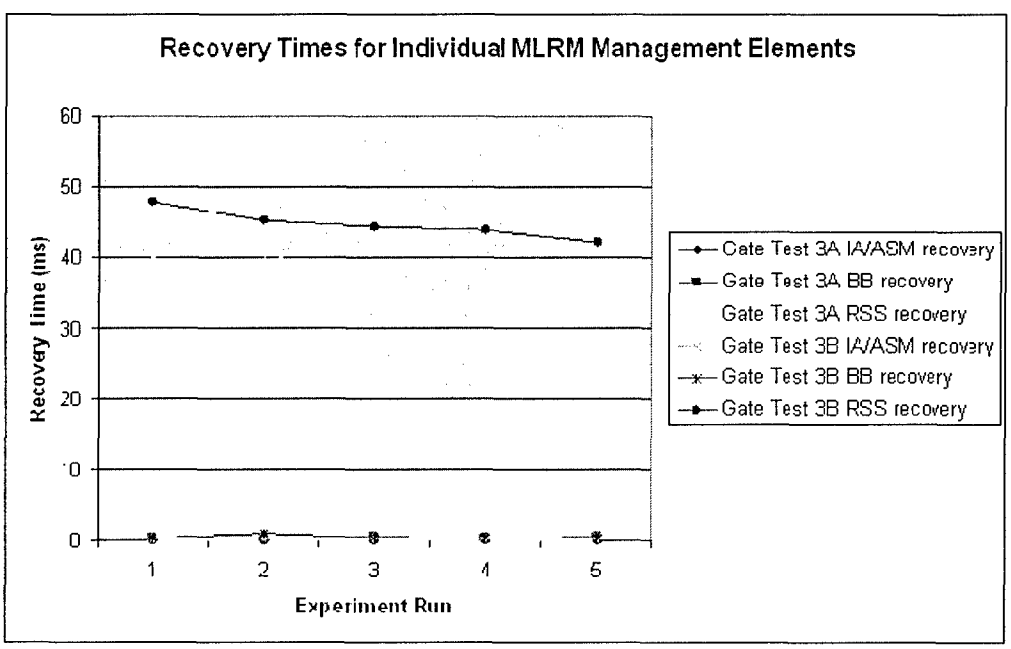

Figure 5: Recovery Time for MLRM Elements

Since we are using Spread's group communication to pass around the failure detection (through its group membership consensus protocol), each individual element, i.e., the IA/ASM-G, BB, and RSS, has a time at which it knows a failure has occurred and starts recovering. By comparing those times from 3A to 3B for each individual element, we can determine whether the difference in recovery times is in the time the detection gets propagated or in the time the elements take to recover.

Using Table 3 and Figure 5, we can make two observations:

1. The actual recovery time of an individual MLRM management element, once it has received the failure event, is very similar from Gate Test 3A to 3B.

2. The RSS recovery dominates total recovery in both Gate 3A and 3B, taking approximately 40 ms versus less than or equal to one ms for the other MLRM management elements.

Observation number 2 is significant, since if something other than the RSS is the first MLRM element to receive the failure event, the time to propagate that failure event to the RSS will be *added to* the total MLRM recovery time. On the other hand, if the RSS is the first element to receive the failure event, the time to propagate the failure event to the other elements and their subsequent recovery will be *concurrent with* the RSS recovery.

Table 4: Gate Test Failure Propagation Statistics

Gate Test	Smallest time to propagate a failure event in an experiment run (ms)	Largest time to propagate A failure event in an Experiment run (ms)	Average(ms)	Standard Deviation (ms)
3A	1.8	50.5	18.3	22.7
3B	1.6	35.5	13.3	13.2

In looking at the times to propagate the failure events during the experimental runs for Gate Test 3A and 3B, shown in Table 4, we notice more variance in Gate Test 3A than in Gate Test 3B. This explains, in part, why the average, maximum, and standard deviation are larger in Gate Test 3A than in Gate Test 3B. Another factor to consider is which element receives the failure event first, because unless the element with the largest recovery time (the RSS in all the experimental runs) is the first element to detect the failure, then the difference in receiving the detection event gets included in the MLRM recovery time.

Table 5 and Table 6 show why the MLRM recovery time appears to decrease in Gate Test 3B from 3A. In 3A, shown in Table 5, the IA/ASM is the first MLRM management element to receive the failure detection event (the time we collect as the MLRM failure detection time). In two cases, it takes tens of milliseconds to propagate to the RSS and then the RSS begins its recovery, which takes about 40 milliseconds. The propagation time is added into the recovery time for the MLRM total recovery time.

In contrast, in 3B, shown in Table 6, the two longest times taken to propagate the failure detection event were in experiments in which the RSS received the event first. Even though the propagation in those two cases takes tens of milliseconds, it is happening concurrently with the RSS recovery (approximately 40 ms). Once the IA/ASM and BB receive the failure detection event, each of them recovers in one ms or less. So the faster 3B recovery time is an artifact of the RSS being notified before any of the more quickly recovering elements.

It is a reasonable question to ask why different elements receive the failure event first and why there is a variance in failure propagation between experiment runs. Two possibilities are:

1. Uncertainties introduced by the Spread group consensus protocol, which we use to propagate the failure event and which is based on a token passing scheme. Differences in where the token starts, whether there are messages waiting to be delivered, and what the group consensus algorithm is doing when the failure occurs can introduce variability.

2. Uncertainties introduced by using the Emulab testbed and by running the experiments over a long span of time. Each experiment took hours to run at the Emulab so the time between running the first Gate Test 3A experiment and the last Gate Test 3B experiment was days. Although we set up each experiment in the Emulab the same way, it is a shared testbed and we don't have complete control over the infrastructure to eliminate all variables introduced by testbed configuration, load, and other factors.

Notice that in the Gate Test 3A and 3B experiments reproduced in the ISISlab, reported in Section 3.2.4.2, the difference reported here disappears.

Table 5: Failure Propagation Times for GT3A

Run	First detection time (ms)	Last propagation time (ms)	Difference	What detects first
1	112.7	163.2	50.5	IA/ASM
2	115.4	149.7	34.3	IA/ASM
3	120.2	122.0	1.8	IA/ASM
4	137.6	139.8	2.2	IA/ASM
5	123.8	126.7	2.9	IA/ASM

Table 6: Failure Propagation Times for GT3B

Run	First detection time (ms)	Last propagation time (ms)	Difference	What detects first
1	239.9	275.4	35.5	RSS
2	237.6	252.0	14.4	RSS
3	137.0	144.2	7.2	IA/ASM
4	51.5	59.5	8.0	IA/ASM
5	116.3	117.9	1.6	IA/ASM

3.2.4 Gate Test 3 Icing - Going Above and Beyond the GT 3 Requirements

In addition to meeting the letter of the law of Gate Test 3, we undertook several activities that went above and beyond the definition of the Gate Test. This section describes each of these.

First, we created ARMS capabilities to reconstitute replicas after a failure. This was a significant undertaking above and beyond the Gate Test, and a research result in its own right. As of the start of ARMS phase 2, there was no software for providing fault tolerance for component applications and no capabilities for dynamically deploying components. We had to develop the concepts for extending fault tolerance (initially developed for pure client-server object applications) to work with components (with their peer-to-peer and multi-tiered semantics). Since a fault tolerance solution that would tolerate faults, but not be able to get back up to a desired level of redundancy is incomplete, we also needed to develop the concepts and software for dynamic component deployment within a replication framework. We report the results of that research and development below.

Second, we reproduced the Gate Test 3 experiments on ISISlab, hosted at Vanderbilt University. The Gate Test 3 definition specified using the Emulab testbed, hosted at the University of Utah and all the results reported above are from experiments run at the Emulab. However, ISISlab, a testbed emerging at Vanderbilt, includes hardware more representative of the PoR. We ran the Gate Test 3 experiments at the ISISlab and the results we report in Section 3.2.4.2 are more representative of what could be expected in the PoR environment. In doing this, we also helped mature the ISISlab testbed and its support for experiments like Gate Test 3.

Third, not satisfied with simply meeting the Gate Test 3A requirements with the MySQL database recovery, we further tuned the database failure recovery mechanisms to see whether we could get it well below the requirements, thereby making it more suitable for real-time recovery. Our efforts paid off and we significantly exceeded the PoR recovery requirement. The results are reported below in Section 3.2.4.3.

3.2.4.1 Replica Reconstitution

Although not explicitly part of the defined Gate Test 3, we undertook to restore the level of fault tolerance after a fault to its pre-fault level. This is an obvious piece of icing toward having a fully fault tolerant capability, since to not do so would mean that the system would be less fault tolerant after recovery than it was before and periodic recurring failures would prematurely lead to complete failure.

Replica reconstitution means that after a fault (or multiple faults), we redeploy new replicas to get back up to a desired level of fault tolerance (i.e., the same level of readily available redundancy as existed before the faults). This includes deploying new replicas and loading them with the state of existing replicas.

Developing and experimenting with replica reconstitution presented some challenges, including the following:

- Prior to ARMS Phase 2, there was no dynamic deployment capability in the CIAO component middleware being used in ARMS. Since many of the MLRM elements being recovered were implemented as CIAO components, we had to either work around the component middleware or develop dynamic deployment capabilities for it. We did both in parallel, so that we could push the fault tolerance and dynamic component deployment research and development forward concurrently.

- Similarly, prior to ARMS Phase 2, the FT code bases that existed did not handle replicating components. As part of Gate Test 3, we had to design and develop capabilities to replicate components (handling their novel peer-to-peer, multi-tiered semantics and deployment infrastructure that hung around at runtime). As part of the replica reconstitution icing, we had to design and develop ways that the component deployment and fault tolerance infrastructure could cooperate to not only deploy a new component, but then to have it become a replica (join the right group) and synchronize its state with the surviving replicas.

We developed these capabilities and, as part of the Gate Test experiments executed above, also ensured that we could restore the replication to the desired level and measured the speed of replica reconstitution. Deployment of a new replica takes just a few seconds. During all but a tiny part of that time, the MLRM functionality (i.e., the surviving MLRM replicas) is up and running and fully functional. There is a brief interruption (on the order of a few 10s of milliseconds) when the MLRM synchs up its state with the new replicas. The timeline in Figure 6 shows the replica reconstitution process and the brief interruption of MLRM functionality.

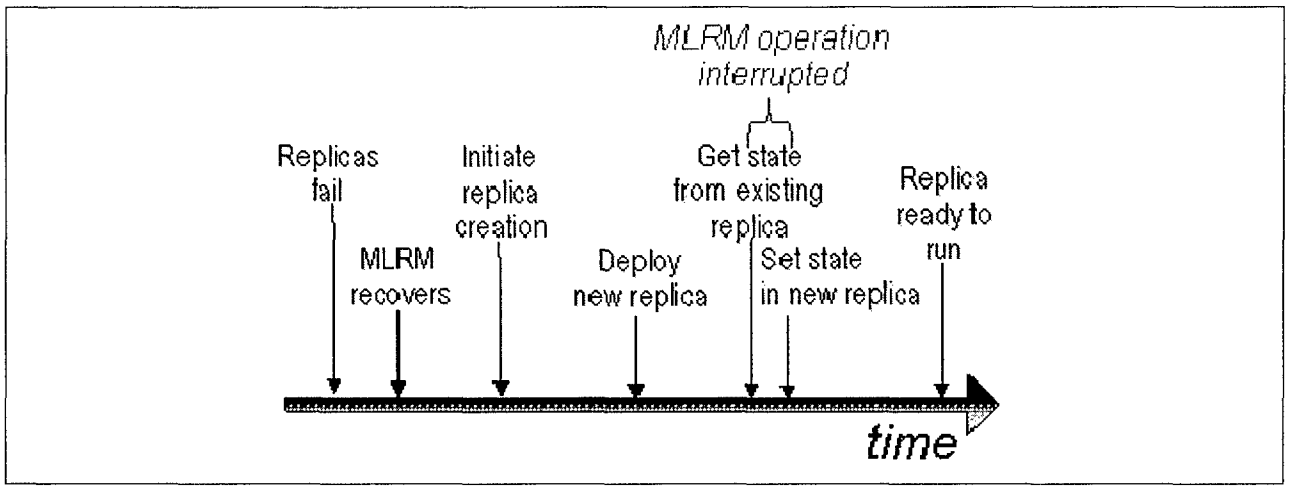

Figure 6:Timeline for reconstituting a Replica

Figure 7 shows the total time to deploy new replicas of the MLRM management element from a representative five experimental runs on Emulab.

For the IA/ASM element, the time to deploy a new replica component consists mainly of the time to deploy new CIAO components. With the BB and RSS, it is the time associated with starting new Java processes, including starting the JVM, loading classes, and so on. During all but a small amount of this time, the MLRM continues to operate. Figure 8 shows the total amount of MLRM downtime (during state synchronization) during the five representative runs. In each experiment the downtime is less than 60ms.

The downtime is the time that a primary is packaging up its state to send to the new replica. The state is primarily the MLRM element's state, but also includes a small amount of middleware state. The state (and therefore the downtime) can grow or shrink based on what the element is doing, e.g., how many strings have been deployed. The RSS downtime also includes some overhead of translating between Java, which the RSS is written in, and C++, which the fault tolerance middleware is written in. The BB downtime is nearly zero because the BB manager element is stateless; all of the BB state is contained in the commercial BB DB, which has non real-time startup and state transfer characteristics.

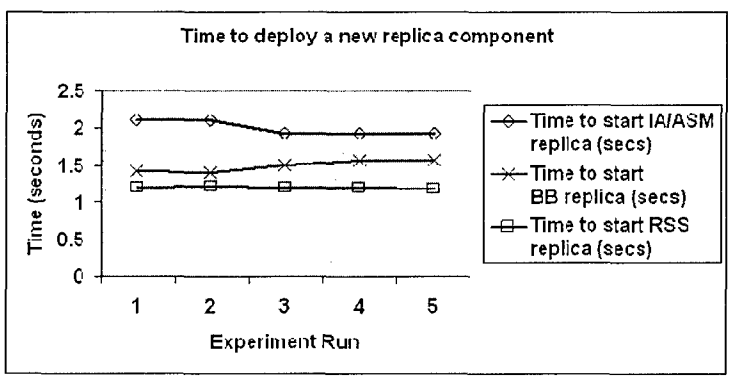

Figure 7: Time to deploy a new replica component on Emulab

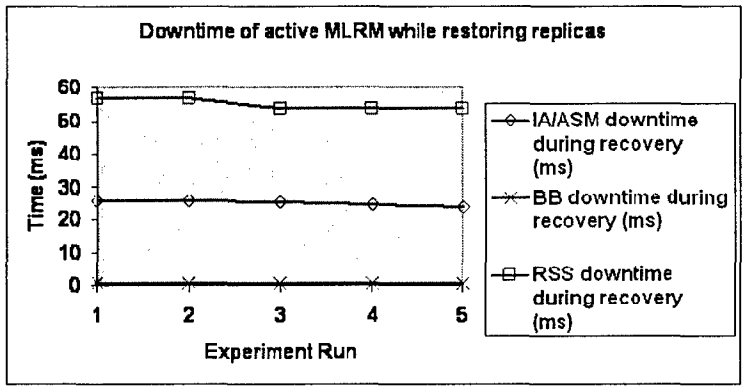

Figure 8: Downtime of active MLRM while restoring replicas

3.2.4.2 Results from Running Gate Test 3 Experiments on ISISlab

We reproduced the Gate Test experiments on the ISISlab testbed at Vanderbilt University. This section describes Gate Test 3A results on ISISlab with the same code that was used to conduct the Gate Test 3A experiments on Emulab, so that we can compare them to those we performed on the Emulab and reported in Section 3.2.3.2. The ISISlab hosts are significantly faster than those on Emulab so once a failure is detected carrying out the recovery logic is done much more quickly. (Below we show full Gate Test 3A Gate Test experiments on ISISlab, which were conducted without the tuned database.)

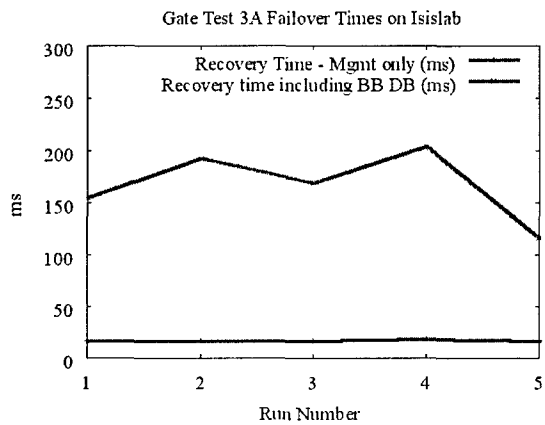

Figure 9: GT 3A Failover Times on ISISlab

Table 7: Results from Five GT3A runs on ISISlab

MLRM Management	MLRM Management	MLRM including BB DB
Average recovery time (ms)	17.1	167.9
Minimum recovery time (ms)	16.7	116.9
Maximum recovery time (ms)	18.2	204.8
Standard Deviation (ms)	0.5	30.8

Figure 9 and Table 7 show that we are able to recover MLRM management functionality in a fraction of the PoR recovery requirement time, more than 17x faster than the requirement time on average. Even including the commercial database technology, we are more than 1.75x faster than the requirement time on average.

Compared to the Gate Test 3A results executed on Emulab, MLRM management functionality recovers 3.5x faster on average (4.9x better than the worst case, with less than one-fourth the standard deviation). The MLRM including the database recovers 1.27x faster on ISISlab than on Emulab on average (1.38x better than the worst case, with a 40% lower standard deviation).

3.2.4.3 Results from Tuning the Bandwidth Broker Database Recovery

It is quite clear from the Emulab and 3A ISISlab results that the MySQL database takes the longest to recover from a failure. In order to improve this recovery time, we tuned the configuration of the MySQL cluster and optimized the communication paths with a small source modification. The result is a much quicker recovery after a failure.

The numbers in Table 8 and values shown in Figure 10 are from a 3A-like scenario. Cluster DB instances are running on two pools and the connectivity between the pools is taken down. Since there is no MLRM running to detect the failure we report only the times between the fault injection and the recovery of the DB shown by a successful query.

With the tuning, the database is able to recover more than 40% quicker *from the time of failure*, with a 60% reduction in standard deviation. As evident below in the re-execution of the Gate Test 3 results on the ISISlab with the tuned database, this has a significant impact on the time of recovery of the MLRM *after detection of a fault.*

Table 8: Database Recovery Statistics, Original and Optimized on ISISlab

	Original	Optimized
Average recovery time (ms)	281.2	157.3
Minimum recovery time (ms)	226.4	141.1
Maximum recovery time (ms)	322.9	179.8
Standard Deviation (ms)	34.4	13.4

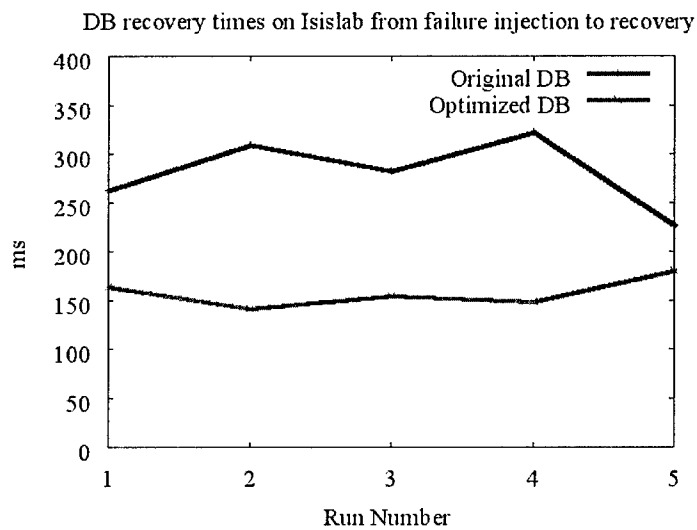

Figure 10: Time from Fault Injection to DB Recovery on ISISlab

Table 9: 3A Results on ISISlab with a tuned BB DB

	MLRM Management	MLRM Including BB DB
Average recovery time (ms)	17.2	31.9
Minimum recovery time (ms)	16.8	16.8
Maximum recovery time (ms)	17.5	60.6
Standard Deviation (ms)	0.3	16.7

3.2.5 Gate Test 3 Experiments on ISISlab with a Tuned BB DB

We reran the Gate Test 3 experiments on the ISISlab after tuning the database. These results are beyond the letter of the law of the GT3 definition because they are on the ISISlab instead of the Emulab, and because they include the database component not used by the PoR. However, as mentioned above, the ISISlab is more representative of the PoR hardware and we wanted to experiment with whether we could vastly exceed the PoR recovery requirement time even with elements above and beyond those required by the PoR.

So, these sets of experiments show how well the full MLRM top level system, with a tuned database, can recover from single and cascading failures on ISISlab Blade hardware, with a goal of exceeding the Gate Test 3 recovery requirement. To do so is well above and beyond the definition of the Gate Test and represents a significant research and development accomplishment, as well as setting the stage for transition of this technology to the PoR.

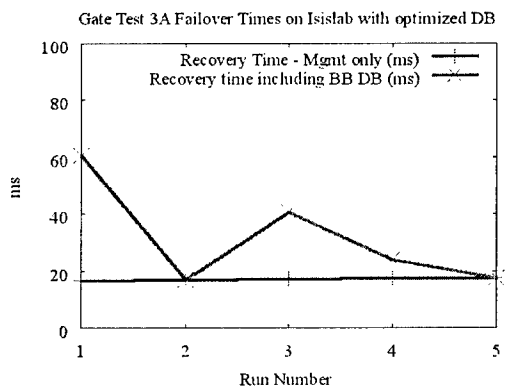

Figure 11: 3A Failover Times on ISISlab with an optimized DB

Table 10: 3A Results on ISISlab with a tuned BB DB

	MLRM Management	MLRM Including BB DB
Average recovery time (ms)	17.2	31.9
Minimum recovery time (ms)	16.8	16.8
Maximum recovery time (ms)	17.5	60.6
Standard Deviation (ms)	0.3	16.7

3.2.5.1 Gate Test 3A Executed on ISISlab with a Tuned BB DB

Notice that in Table 10 and Figure 11 the MLRM management numbers are very similar to those reported earlier with the untuned database, as we would expect. However, now with the tuned database, the maximum recovery time is 3.4x better than with the untuned database, and nearly 5x faster than the Gate Test 3A recovery requirement time. On average, ***recovery time of the full MLRM system with the tuned database is nearly 10x the PoR recovery requirement time***.

The ARMS real-time fault tolerance capabilities, exemplified by the MLRM management recovery time, still outperform the capabilities of the commercial database solution with nearly 2x faster recovery time on average and over 50x more predictability (as measured by the standard deviation), as would be expected. However, as a result of ARMS research and engineering, these results show that ARMS fault tolerance can provide real-time fault tolerance for more RM functionality than the baseline system in well under the PoR recovery requirement time.

3.2.5.2 Gate Test 3B Executed on ISISlab with a Tuned BB DB

Again, Table 11 and Figure 12 shows the ability to reproduce the ability to recover from cascading failures, exceeding the requirements of the PoR, on the ISISlab hardware, which is similar to that of the PoR. MLRM recovery is very fast. ***Even though Gate Test 3B is not designed to be compared to the PoR recovery requirement time, it compares very favorably, with recovery (from cascading failures) of management functionality more than 16x faster than the recovery requirement time on average, and recovery of full MLRM including the BB DB more than 5x faster than the recovery requirement time on average.***

Table 11: Statistics aon 3B runs on ISISlab with a tuned BB DB

	MLRM Management	MLRM including BB DB
Average recovery time (ms)	18.85	59.4
Minimum recovery time (ms)	18.37	22.9
Maximum recovery time (ms)	19.48	84.1
Standard Deviation (ms)	0.38	20.2

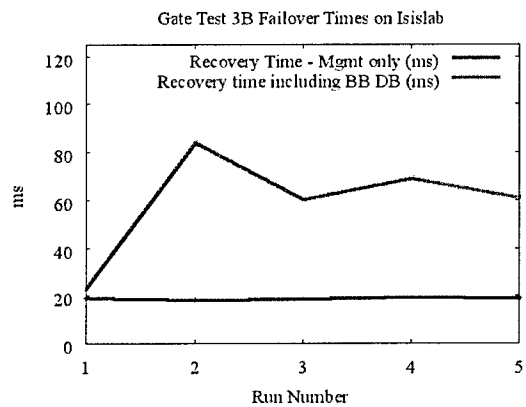

Figure 12: 3B Failover Times on ISISlab

3.2.6 Conclusions

The results that we have presented in this section indicate that our efforts for ARMS Gate Test 3 have been extremely successful in not only satisfying the requirements laid out for the Gate Test, but also *vastly exceeding them*, in the form of collected metrics and in the results that we have produced above and beyond the Gate Test 3 letter of the law.

Looking first at the *letter of the law* requirements of Gate Test 3, we were able to make the MLRM functionality fault tolerant, as required by Gate Test 3A, and to make it handle cascading failures, as required by Gate Test 3B. In doing so, we met the PoR recovery requirement time with functionality (including the Bandwidth Broker database) beyond that of the comparable PoR system and significantly exceeded (by nearly 5x) the PoR's recovery requirement time with MLRM management elements comparable to those in the PoR system, all on Emulab hardware, which is lower performance than the PoR hardware.

In addition, we replicated the experiments on hardware more representative of the PoR hardware (on the ISISlab) and further tuned the additional MLRM elements (i.e., the BB DB). Upon doing so, we beat the PoR recovery requirement time by over 17x with the apples-to-apples comparable MLRM management elements and by nearly 10x with the additional BB DB elements. Furthermore, not only could we recover from cascading failures, we could do so more than 16x times faster than the PoR recovery requirement time with the MLRM elements and more than 5x faster than the PoR recovery requirement time including the BB DB, *even though the cascading failure scenario was not meant to be compared to the PoR recovery requirement time.*

The ARMS fault tolerance technology exhibits the characteristics needed for strict real-time environments such as the PoR in both its rapid recovery and its highly predictable, low variance recovery. Although we were able to tune the COTS database technology to recover more rapidly and thereby make it more suitable for real-time applications, it still exhibits the higher variance of a non real-time solution.

We also produced *icing* in the form of increased capabilities above and beyond those required by Gate Test 3 such as: reconstituting replicas by returning to a desired level of fault tolerance, which required significant research in dynamic component deployment and fault tolerance for component models; and running tests on ISISlab with an optimized database.

To achieve Gate Test 3, we advanced the state of the art in fault tolerance technologies, including the following:

- Fault tolerance for components. Previous fault tolerance solutions worked with objects or databases.
- Fault tolerance for multi-tiered applications and peer-to-peer applications. Previous FT solutions worked only for single-tiered applications (i.e., replicated elements were pure servers) and round-trip client-server communication.
- Co-existence of group communication and non-group communication. Previous FT solutions required group communication, if used, to be pervasive.
- Mixed-mode fault tolerance, i.e., active, passive, and transactional database coexisting.
- Multi-ORB (TAO/CIAO and JacORB) and multi-language solutions, C++ and Java

The following section describes these R&D results in more detail.

3.3 Fault Tolerance Research and Development Results

Moving from a non-FT MLRM to an FT MLRM required solutions to a number of technical challenges. We first introduce the fault model under which we have been designing the system. The following sections then discuss some of the research we have undertaken as part of our ARMS work. Following this we discuss a number of FT challenges and the solutions we developed for ARMS, both in the context of the Gate Test and also in a broader fault tolerance domain. These include solutions to dynamically supporting components and their multi-tiered and peer-to-peer interaction models, supporting multiple languages, supporting multiple replication schemes, and supporting efficient communication when replication is being used. We then discuss some of the development necessary to enable the gate test and highlight experimentation showing the cost of using our solution relative to a non-fault-tolerant solution and lessons learned. Finally, we also note future directions and ideas for further work.

3.3.1 Fault Model

A fault model describes the types of failures we expect our system to have to deal with. By being specific about our fault model we both enable simpler solutions when arbitrarily malicious failures are not a concern and also make clear the types of failures the system is designed to deal with.

In designing our FT solution we assume that all faults are fail-stop at the process level, i.e., when an application process fails it stops communicating and does not obstruct the normal functioning of other unrelated applications. Network and host failures can be seen as a collection of process failures on the network or host that has failed. Some examples of failures that we will tolerate include power being disrupted to a host, an application crashing, or a data center being destroyed. Some examples of failures that we do not currently tolerate are network partition recovery and general Byzantine [11] failures. When a network splits (partitions), perhaps due to a network failure, leaving two groups of replicas that move forward independently we assume that they will never join together again, greatly simplifying the system. Malicious, or Byzantine, failures are where a process may intentionally attempt to deceive other members or misrepresent data. Tolerating Byzantine failures requires many constraints on the system and also requires considerable extra resources. By dealing exclusively with crash failures we are able to support many more types of applications with fewer resources.

3.3.2 Challenges in Providing Fault Tolerance in DRE Systems

DRE systems provide unique challenges to using many existing fault tolerance implementations because of the scale, real-time requirements, dynamic configurations, and calling semantics typical of DRE systems. This section describes four particular challenges with applying existing fault tolerance solutions to the needs of DRE systems:

- Communicating with replicas in large scale, mixed mode systems
- Handling dynamic system reconfigurations
- Handling peer-to-peer communications and replicated clients and servers.
- Supporting a multi-paradigm, multi-language environment

3.3.2.1 Communication with groups of replicas

Fault tolerance is commonly provided using replication, which requires a means to communicate with groups of replicas. A common approach is the use of a group communication (GC) system (GCS), which ensures consistency between replicas and between replicas and their non-replicated clients or servers. DRE systems provide several challenges for using GC. They contain large numbers of elements with varying fault tolerance requirements. Some elements will have stringent real-time requirements. This means that GC might not be needed, or even acceptable, in many places in the system. The following paragraphs describe approaches to group communication and its applicability to DRE systems.

Pervasive GC. Some approaches [1] use GC for communication throughout the entire system. This approach provides strict guarantees and ensures that interactions between applications and replicas are always done in the correct manner. It can, however, limit the scalability of resulting systems and add extra overhead associated with group communication on the communication of elements that don't need GC. In very large DRE systems, such as the one in which the MLRM runs, non-replica communication can be the more common case and using GC everywhere can severely impact performance.

Pervasive GC is problematic in component-oriented systems due to features of component deployment. The deployment of components both when new applications are deployed and when additional replicas are needed (e.g., to replace replicas that have failed) is done using a CORBA-based deployment framework. The messages related to the deployment of a CCM-based replica are of concern only to the new replica, yet the use of pervasive GC results in deployment messages going to existing replicas (which were previously deployed). Thus, replicating components requires the coexistence of non-group communications (during the deployment of a new replicated component) and group communications (once all replicas have been fully deployed).

Gateways. Other systems [3, 4] make use of gateways on the client-side that change interactions into GC messages. This limits group communication to communication with replicas and provides the option to use non-GC communication paths where necessary. It is therefore useful in applications that need group communication for replicas but cannot afford to use group communication everywhere, especially in applications with relatively small numbers of replicated elements. The gateway approach does come with tradeoffs, however. First, it is less transparent than the pure GC approach because the gateway itself has a reference that has to be explicitly called. Second, gateways typically introduce extra overhead (since messages need to traverse extra process boundaries before reaching their final destination) and extra elements that need to be made fault tolerant to avoid single points of failure. Other gateway-like strategies [5,6] have also been explored, similar to the "fault-tolerance domain" specified in FT-CORBA.

Other projects [7] take a hybrid approach where GC is only used to communicate between replicas and not to get messages to the replicas. This places the gateway functionality on the server-side of a client-server interaction, which limits the interactions between replicated clients and replicated servers but has implications for replicating both clients and servers at the same time. It introduces the possibility that lost messages may need to be dealt with at the application level as they cannot use the guarantees provided by the GC system.

ORB-provided transports. Some service-based approaches [8] completely remove GC from the fault-tolerance infrastructure and use ORB-provided transports instead, which limits them to using passive replication.

3.3.2.2 Configuring FT Solutions

A recurring problem with using GC in dynamic systems like DRE systems is keeping track of groups, replicas, their references, and their supporting infrastructure as elements come and go during the life of a large DRE system. Many existing fault tolerance solutions make use of static configuration files or environment variables [1,3]. The DRE systems that we are working with are highly dynamic, with elements and replicated groups that can come and go and need to make runtime decisions about things such as fault tolerance strategy, level of replication, and replica placement. Static configuration strategies lack the flexibility needed to handle these runtime dynamics. Eternal [9] supports dynamic fault tolerance configurations. Greater flexibility is also available in some agent-based systems [10] but for more common non-agent infrastructures adding additional FT elements to a running system is not common.

3.3.2.3 Replicated Client and Servers, Peer-to-Peer Interactions, and Multi-Tiered Replication

Support for replicated servers is ubiquitous in fault tolerance replication solutions, whereas support for replicated clients is not as common. Many CORBA-based fault tolerant solutions concentrate on *single-tier* replication semantics, in which an unreplicated client calls a replicated server, which then returns a reply to the client without making additional calls. Multi-tiered or peer-to-peer invocations are possible but the FT-CORBA standard [11] does not provide sufficient guarantees or infrastructure to ensure that failures, especially on the client-side, during these invocations can be recovered from. A similar situation exists in some service-based approaches [8,12] where peer-to-peer interactions are possible but care must be taken by developers to make use of the functionality.

In contrast, component-oriented applications exhibit peer-to-peer communication patterns, in which components can be clients, servers, or even both simultaneously. Many emerging DRE systems are developed based on component models and exhibit peer-to-peer calling structure. Because of this, fault tolerance strategies and solutions based on strict server replication are of limited applicability.

Since components can be both clients and servers, frequently component-oriented DRE systems have chains of nested calls, in which a client calls (or sends an event to) a server, which in turn calls another server, and so on. This leads to a need to consider replication of multiple tiers of servers. Research into supporting fault-tolerance in multi-tiered applications is still ongoing. Some of the most promising recent work has concentrated on *two-tier replication*, specifically addressing applications consisting of a non-replicated client, a replicated server, and a replicated database [13].

General, unrestricted calling patterns, such as asynchronous calls, nested client-server calls, and even callbacks (where clients also act as servers and can have messages arrive via the callback mechanism while replies from sequential request-reply messages are pending), present tremendous challenges for fault tolerance solutions. This is partially due to the need for fault tolerance to maintain message ordering, reliable delivery, and state consistency, which is harder to do in asynchronous, multi-threaded, and unconstrained calling patterns. It is also due to the fact that the semantics of such calling patterns in the face of replication are more difficult to define.

3.3.2.4 Supporting a Multi-Paradigm, Multi-Language Environment

The MLRM environment is not a simple homogeneous one. It contains C++ components intermixed with Java and C++ CORBA objects as well as different ORBs, aspects of which need to be made fault-tolerant and interoperable. This heterogeneity requires a fault tolerance solution that can support components and objects programmed in both Java and C++. Furthermore, the desire to have a transparent solution (to the degree possible) is often in conflict with the desire to be portable and efficient across different implementations and platforms.

3.3.3 Fault Tolerance Solutions to the Challenges for DRE Systems

In this section, we describe three new fault tolerance advances that we developed under the ARMS program. First, we describe a *Replica Communicator (RC)* that enables the seamless and transparent coexistence of group communication and non-group communication while providing guarantees essential for consistent replicas. Next, we describe a self-configuration layer for the RC that enables dynamic auto-discovery of new applications and replicas. We then describe an approach to and implementation of duplicate message management for both the client- and server-side message handling code in order to deal with peer-to-peer interactions. Finally we discuss the need to support heterogeneity which is an essential component of the three advances.

3.3.3.1 The Replica Communicator

In order to provide the GC underpinnings necessary for maintaining consistent replicas while at the same time limiting unnecessary resource utilization and not disturbing the delicate tuning necessary for real-time applications, we needed a way to limit the use of group communication to those places in which it was absolutely necessary. Analysis of various replication schemes shows that the only place where GC is necessary is when interacting with a replica. That is, only replicas and those components that interact with them need the guarantees provided by group communication. Other applications can use TCP without having to accept the consequences of using GC, whose benefits are not needed in their case.

There are several advantages to limiting the use of GC to only those places in which it is needed. The first reason is that GC introduces a certain amount of extra latency, overhead, and message traffic that is undesirable in the non-replica case and, in fact, can jeopardize real-time requirements. Second, many off-the-shelf GC packages, such as Spread [14], have built-in limits on their scalability and simply do not work with the large-scale DRE systems that we are targeting. Finally, many of the components of our targeted DRE systems are developed independently. Since the non-replicated case is the prevalent one (most components are not replicated), retrofitting these components onto GC, with the subsequent testing and verification, would be a tremendous extra added effort for no perceived benefit.

Therefore, we developed a new capability, called a Replica Communicator, with the following features:

- The RC supports the seamless co-existence of mixed mode communications, i.e., group communication and non-group communication.
- It introduces no new elements in the system.
- It can be implemented in a manner transparent to applications.

The RC can be seen as the introduction of a new *role* in an application, along with the corresponding code and functionality to support it. That is, the application now has three communication patterns, illustrated in Figure 13:

1. Replicas that only communicate with other replicas, which use GC
2. Non-replicas that only communicate with other non-replicas, which use TCP
3. Non-replicas that communicate with non-replicas or replicas, and use an RC to route the communication along the proper protocol.

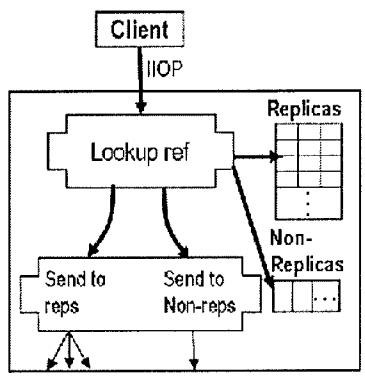

Figure 13: Generalized Pattern of the Replica Communicator

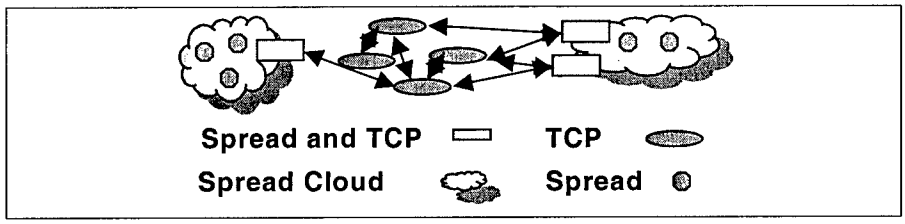

Figure 14: Coexistence of group communication (Spread) and non-group communication, where elements at the edge of the interaction communicate via both transports.

An abstract view of the RC is illustrated in Figure 14. Its basic functionality consists of the following pieces:

- Interception of client calls
- A lookup table to hold references to replicas and to non-replicas
- A decision branch that determines whether a call is destined for a non-replica or a replica and treats it accordingly
- A means to send a message to all replicas, e.g., multicast, a loop over all replica references, or via GCS
- A default behavior, treating a message by default as one of the branches
- A configuration interface to add references to new servers or to new replicas (to an existing group), or to remove a replica (if it has failed)

Documented in the above pattern, the RC can be realized with multiple implementations. Application specific implementations can be made in the application logic itself, using aspect-oriented programming or component assembly to insert the RC transparently into the path of client calls. It can also be realized using standardized insertion points, such as library interpositioning to hook into the system at the system-call level or using the CORBA-standard Extensible Transport Framework (ETF). Transparent insertion is highly desirable from an application-developer's point of view since it makes fault tolerance easier to integrate.

The RC functionality resides in the same process space as the application. This improves over traditional gateway approaches, because it introduces no extra elements into the system. Notice that the RC does not need to be made fault tolerant, since it is not a replica.

We have realized a prototype of the RC pattern in the MLRM based on the MEAD framework [1] and its system call interception layer, as illustrated in Figure 15. CORBA calls are intercepted by MEAD, which is added to applications at execution time through dynamic loading of libraries. The RC code maintains a lookup table associating IP addresses and port numbers with the appropriate transport and group name if GC is used. The default transport is TCP; if there is no entry in the lookup table, the destination is assumed to be a non-replicated entity. For replicated entities, the RC sends the message using the Spread GCS, which provides totally-ordered reliable multicasting. For replies, the RC remembers the transport used for the call, and returns the reply in the same manner.

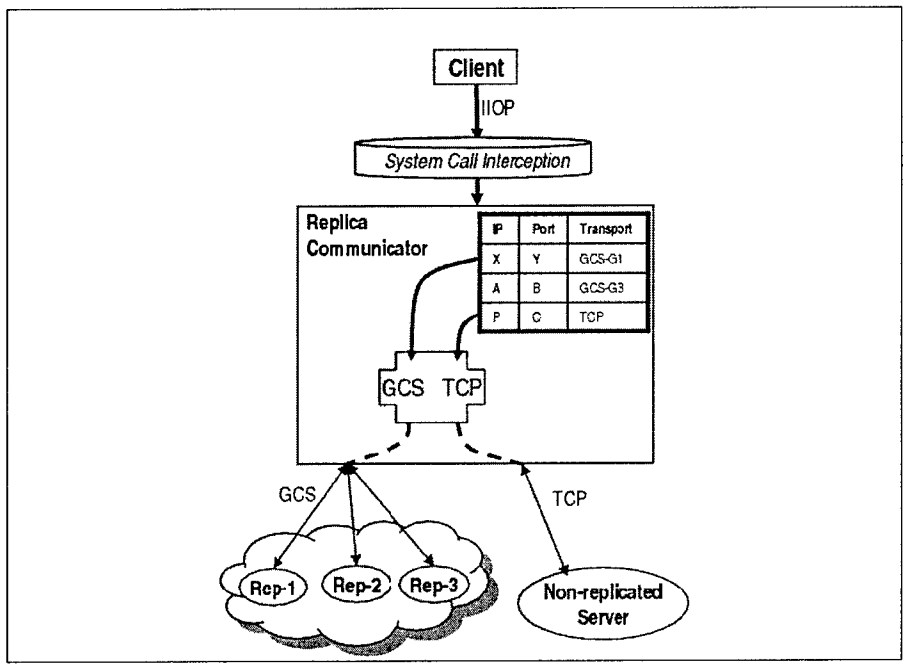

Figure 15: The Replica Communicator Instantiated at the System Call Layer

The Replica Communicator was crucial for resolving the problem outlined in Section 3.3.2.1, namely that the CCM deployment infrastructure needs a way to communicate with exactly one replica during bootstrapping. We used the RC with our CCM-based active and passive replicas to allow a replica to be bootstrapped while not disturbing the existing replicas.

3.3.3.2 A Self-Configuring Replica Communicator

Populating the table distinguishing GC and TCP endpoints shown in Figure 16 can be done in multiple ways. One way is to set all the values statically at application start-up time using configuration files. However, this leads to static configurations in which groups are defined a priori and supporting dynamic groups and configurations is difficult and error prone. To better support the dynamic characteristics of DRE systems and to simplify configuration and replica component deployment, we developed a self-configuring capability for the RC.

When a GC-using element (i.e., a replica or non-replica RC) is started we have it join a group used solely for distributing reference information. The new element announces itself to the other members of the system (shown by the arrows labeled 1 in Figure 16), which add an entry to their lookup table for the new element. An existing member, in a manner similar to passive replication, responds to this notification with a complete list of system elements in the form of an RC lookup table (the arrow labeled 2). The new element blocks until the start-up information is received, to ensure that the necessary information is available when a connection needs to be established (i.e., when the element makes a call). When an element using the RC pattern attempts to initiate a connection, it might be a call that needs to use GC or one that shouldn't use GC. Since GC-using elements always register and are blocked at start-up until they are finished registering, the RC has all the information it needs to initiate the correct connection. If there is no entry for a given endpoint it means that TCP should be used for that connection.

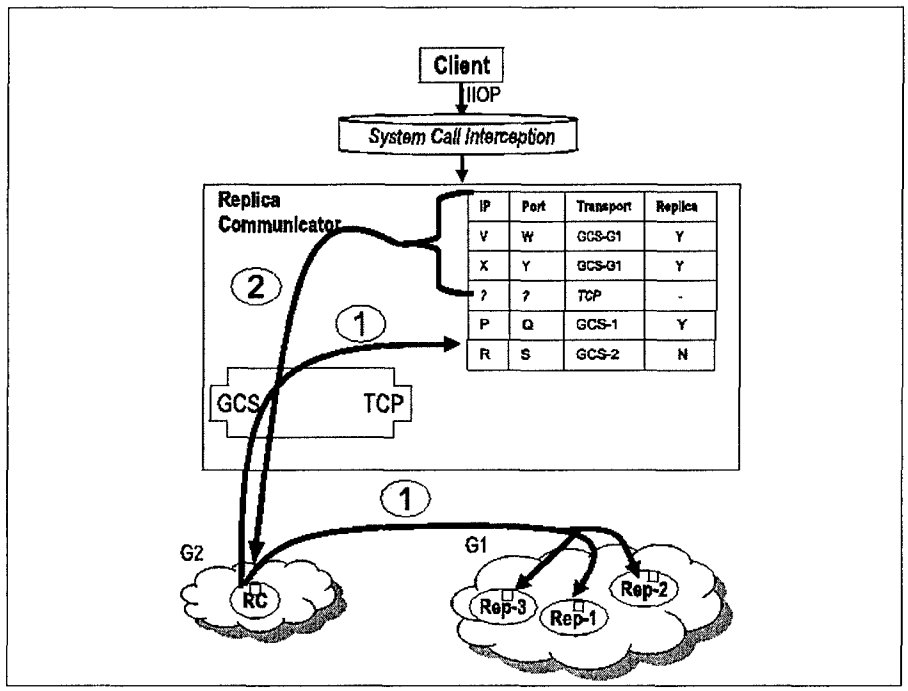

Figure 16: Steps in updating a new RC Reference

One complexity that does not affect users, but needs to be taken into account while developing the self-configuring RC, is that the relationships between elements are not necessarily transitive. Simply because RC1 interacts with replica R via GC and R also interacts with RC2 via GC, this does not mean that RC1 should use GC to interact with RC2. In the case of manual configuration this is handled by having a configuration specific for each application. However, in our automated solution it is necessary to do more than note that a given endpoint can be contacted via a given GC group name. We also need to distinguish the circumstances where GC is necessary and those where it is not. We accomplish this by noting whether a reference refers to a replica or non-replica. Given that interacting with a replica or being a replica are the only two times GC is necessary, an RC knows to use GC when it is interacting with a replica (and TCP elsewhere) and replicas always use GC.

3.3.3.3 Client- and Server-Side Duplicate Management

One step towards a solution for replication in multi-tiered systems is the ability for each side of an interaction to perform both client and server roles, at the same time. This is essential to our support of components and allows nested calls to be made without locking up an entire tier while waiting for a response, which can guarantee consistency, but is very limiting.

Supporting these dual roles has two aspects. The first is the ability to send and receive in a non-blocking way. When an application sends out a request the underlying infrastructure cannot block while waiting for a reply as another request may need to be serviced before the first request will return. The second aspect is the ability to distinguish and suppress duplicate messages from both replicated clients and replicated servers. If this suppression is not performed the applications, which do not know they are replicated, can be confused by receiving the same message multiple times. Solutions that require idempotent operations [15] also solve this problem and can work in multi-tiered situations, but are not satisfactory for use in general DRE systems.

One characteristic necessary to support duplicate management is that messages need to be globally distinguishable, both within an interaction and between multiple interactions. Within an interaction, message IDs are often used to distinguish individual messages from one another. However, when multiple senders independently interact with a shared receiver, it is important to differentiate messages based not only on message ID, but to use a combination of message ID and source. In Figure 17 both A and C use sequence number 1 to send a message to B, but since suppression uses both the sequence number and the sender there is no confusion. An important note here is that when a new replica is integrated with existing replicas it is essential that the message ID aspect of the existing replicas' state is transferred to the new replica. Without this a new replica could have all of its (non-duplicate) messages dropped.

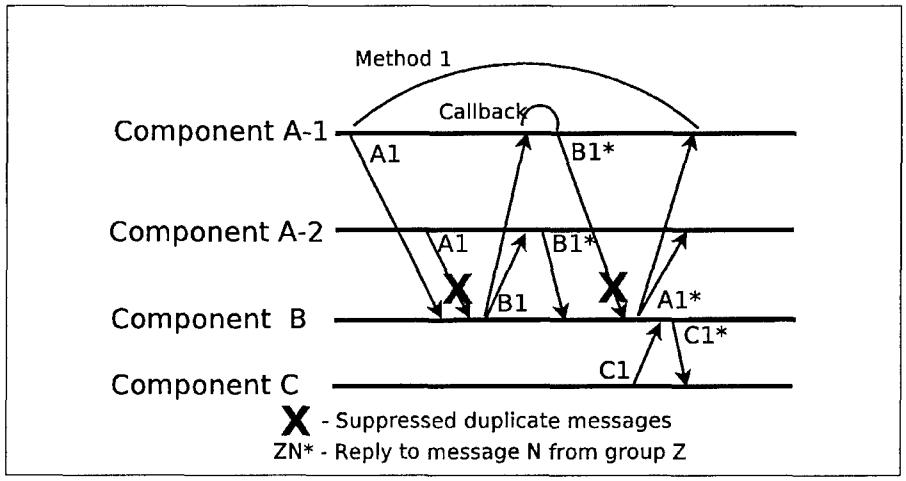

Figure 17: Duplicate Management during Peer-to-Peer Interactions

Our solution enables duplicate management in the highly dynamic situations typical of DRE and component-based software. Requests and replies are dealt with simultaneously and are unaffected by failures that could reset application-level sequence numbers. We replace the ORB supplied request ID with a unique and consistent value for each request or reply and distinguish messages upon receipt using both the ID as well as the sending group. This allows replicas to come and go without introducing any extra messages at the application layer.

3.3.3.4 Supporting a Heterogeneous Environment

Initially MEAD only supported C++ and TAO. It did not support Java, due to difficulties stemming from non-determinism such as garbage collection and threading. In order to be able to run the MLRM we enhanced MEAD so that it worked with the two ORBs used by the MLRM: JacORB, a Java ORB, and CIAO, TAO's CCM implementation. In adding support for JacORB into MEAD we did not remove all sources of non-determinism from Java, but rather dealt with the threading of network IO in such a way that JacORB behaved deterministically when used with active replication and could be used with passive replication.

Whereas C++ applications developed with TAO use non-blocking mechanisms, such as the *select* system call, to wait for responses, the JVM often makes a new thread for each operation and blocks, waiting for the operation to finish. This call style is quite different from what MEAD initially supported and in order to deal with it we added code to the read and write calls that effectively blocked the application while registering a callback. When data was available the waiting thread would be called back and allowed to progress. This preserved the semantics expected by the JVM while allowing us to deliver messages in the ordered manner necessary to make our fault tolerance solution work, all without application knowledge.

Interactions between CORBA components and objects, whether using TAO, CIAO, or JacORB, went quite smoothly compared to the differences encountered supporting Java and C++. Due to the standardization provided by CORBA and a common use of Spread these interactions were not problematic.

3.3.4 Engineering Developments Needed for Gate Test Success

A number of engineering-level developments were necessary to prepare for the Gate Test. This section highlights development-related items that contributed to the Gate Test success: application level state transfer, a special BB fault-tolerance scheme, fault detection, and FT component deployment. It also includes developments made by our Vanderbilt team to support the needs of the GT3 FT solution.

3.3.4.1 Application State Transfer for the MLRM and RSS

The state transfer described in Section 3.2.4.1 described state transfer from a high level as a necessary part of redeploying replicas. While we needed support for state transfer in general in our FT solution each replicated application also needed to support having its state transferred. Unlike the stateless (from an application point of view) BB, both the IA/ASM-G and the RSS are stateful applications and need to transfer their application state when a new replica was started or in the case of the RSS when they were replicated using a passive scheme.

The IA/ASM-G state was straightforward, though ensuring determinism meant that some threaded optimizations were not pursued. When requested by our FT middleware, the IA and ASM-G would gather their application state and return it to the middleware. From there it was combined with the middleware state and transferred to a new replica.

The solution for state transfer in the RSS was more complicated. Whereas the IA/ASM-G only changed state due to network messages (and could thus be actively replicated) the RSS also made use of timers and one-way messages. This meant that it had to be passively replicated but had the additional requirement that timer-based changes needed to be propagated to non-leader replicas on state change, regardless of whether a network event had occurred. In order to support this we added an application-level hook that enabled the application to notify the FT middleware that state had changed and that this state should be share with all replicas.

3.3.4.2 Bandwidth Broker Fault Tolerance Scheme

In Section 3.3.3.1 we discussed how, in order to ensure replica consistency, interactions with a replica need to go over GC. This ensures that all the replicas receive the same messages and that message ordering is preserved. Unfortunately, it assumes that every item interacting with a replica over the network can do so using GC. While this may be possible for many applications there are cases where it is not practical. (Interacting with hardware such as routers and using SNMP are some examples.) The BB provided one such case as it interacted with an "off-the-shelf" database, MySQL.

We needed to replicate the BB but also needed to ensure that the state stored in the DB would not be corrupted due to inconsistent message ordering or a change in the primary replicas. As designed, the BB was split into two parts, a stateless "front-end" Java process that interacted with the rest of the MLRM, and "back-end" DB that saved the BB state but only interacted with the BB front-end. This is shown in Figure 18. In order to not have a single point of failure both of these elements needed to be made fault-tolerant.

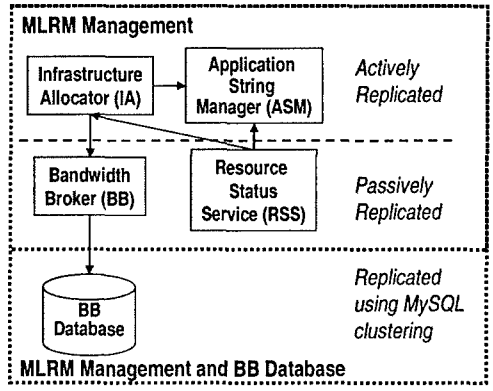

Figure 18: Bandwidth Broker Integration

The front-end was replicated using a custom passive scheme coupled with application-level changes in the BB. As in a traditional passive replication scheme, whenever a message was sent to the front-end it was done so via GC and was received at each member. The non-leaders would buffer the message in case they became the leader and the leader would pass it to the DB. When a response was received by the leader, it would share the response with the requesting client as well as the non-leader BB front-ends. The non-leaders could then remove the request from their buffers. This scheme was optimized from a straight passive scheme in that it did not attempt to transfer state between the leader and non-leaders.

In order to ensure that each message to the DB was processed only once the BB front-end application was modified to add a request-ID to each invocation. This allowed the BB to detect and deal with multiple invocations. This change, coupled with the idempotency of a given message at the DB allowed us to replicate the BB front end.

Our solution for replicating the back-end DB used an off-the-shelf clustering solution modified to detect and recover quickly from failures. By carefully configuring the DB tuning parameters and making a small source code change to allow DB identifiers to be specified at configuration-time rather than coordinated at the time of a failure (saving time and reducing timing variance), we were able to quickly recover from DB failures, as shown in Section 3.2.5.

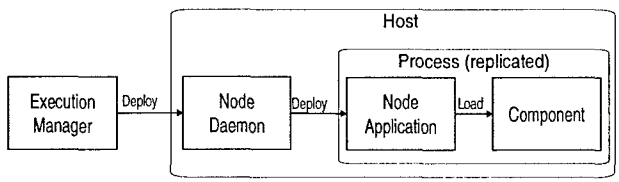

Figure 19: CIAO includes infrastructure elements that are used to deploy components, only some of which need to be replicated.

3.3.4.3 Fault Detection

The first step of being able to deal with a failure is to detect it. We relied on Spread's fault detection capabilities for Gate Test 3, while simultaneously developing the specialized Node Failure Detector (NFD) solution described in Section 4, which provides an independent subsystem for high performance failure detection. In addition to providing messaging guarantees, Spread also detects node failures and issues group membership changes for each group that has a member on the failed node. However, the default configuration that Spread has "out of the box" can take over 5 seconds to detect node failures. We needed faster reaction times from the Spread daemons. Based on previous work that documented Spread tuning [16], we adjusted the timeout parameters in Spread to obtain failure detection times under 200 milliseconds. These changes included increasing the frequency of failure detection messages and decreasing the quiescent time required between the loss of a member and the declaration of a new group membership.

While these tuned timeouts made the node-failure detection time faster, they also made the Spread daemon more susceptible to false-positives caused by latency related to processor scheduling at the operating system level. Our initial testing showed that a high CPU load on a node would cause the Spread daemon to get scheduled less often than required, which in turn caused the other nodes to report the high-load node as failed. The daemon needed to run frequently for very small amounts of time. We solved this problem by making the Spread daemon the highest priority process on every node. Given the default scheduling time-slice on Linux (1 ms), this was sufficient to guarantee that the Spread daemon got a chance to run as often as it needed to.

Once the failures were detected the FT middleware took care of ensuring that the remaining replicas continued to work as expected.

3.3.4.4 Deploying Fault-Tolerant Components

Another challenge due to components is that the deployment architecture of CCM is more complicated than most CORBA 2 solutions. Before a component can be deployed using CIAO, a *Node Daemon (ND)* starts up a *Node Application (NA)*, which acts as a container for new components, as illustrated in Figure 19. The ND makes CORBA calls on the NA, instructing it to start components, which are not present at NA start up time. Note that the components, when instantiated in the NA, need to be replicated, but the NDs should not be.

To illustrate this point, consider an existing fault-tolerant component when a new replica is started. Since MEAD ensures that all messages to and from one replica are seen at every replica, the existing replicas will receive an extra set of bootstrap interactions each time a new replica is started. This will not only confuse the existing replicas, but the responses from the new replica will also confuse the existing NDs. This is one of the motivations for the RC described in 3.3.3.1.

3.3.4.5 Reconciling Objects with Groups and Components with Processes

As part of the GT3 solution we made use of a number of technologies from Vanderbilt, including The ACE ORB (TAO)[17]; the Component Integrated ACE ORB (CIAO); and the Deployment And Configuration Engine (DAnCE) [16],. Out of the box these technologies needed additional development for use in a GT3-like environment. Two of these changes, described below, are reconciling object reference semantics with GC semantics and reconciling procedural and object-oriented models in state synchronization.

Reconciling object reference semantics with GC. In a system using GC all members look alike to the outside world, i.e., they are accessed via a group name. This could result in a single object reference (IOR) that should be available to clients. However, when dealing with relatively transparent replication, enforcing the fact that each replica uses the same common IOR is non-trivial. There is a similarity between GC group names and CORBA interoperable group references (IOGR), but unfortunately the interoperability is between CORBA implementations and not between CORBA and Spread. In order to reconcile these differences, our middleware needs to create exactly the same IOR at each replica. Moreover, when a new replica joins a group we require it to have the same IOR exposed to the GC. In order to enforce this behavior, we modified the portable object adapter within TAO to use the USER_ID and PERSISTENT_ID POA policies. Each set of replicas was given a unique user id corresponding to its group name. This was done in a seamless manner without manual programmatic effort by delegating the job of configuring the policies on the objects (or components) using the DAnCE engine and supplying it with the right set of XML descriptors.

Reconciling process and component models in state synchronization. Our FT framework was initially developed for CORBA-2 based object systems. Newer systems that make use of components have requirements that our middleware was not designed to meet. One area where this occurred was at the interface between our FT middleware and CIAO/TAO/DAnCE.

Within DAnCE, the NodeApplication is a process, which performs the job of an application or component server. We interfaced our FT solution with the NodeApplication process and provided two global functions called *get state* and *set state*. Since the FT middleware cannot differentiate the state of individual components (or objects) we needed to modify CIAO to turn a single call to get state into multiple calls to each component in the process. This is done using the DAnCE's domain application manager, which in turn instructs all the NodeApplication processes to get/set the state during recovery.

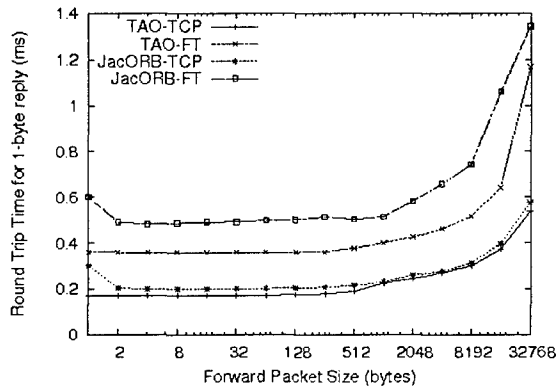

Figure 20: Latency of Transport Mechanisms with 1 Replica

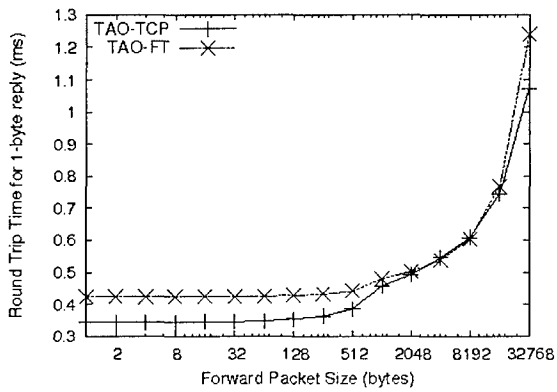

Figure 21: Latency of Transport Mechanisms with 2 Replicas

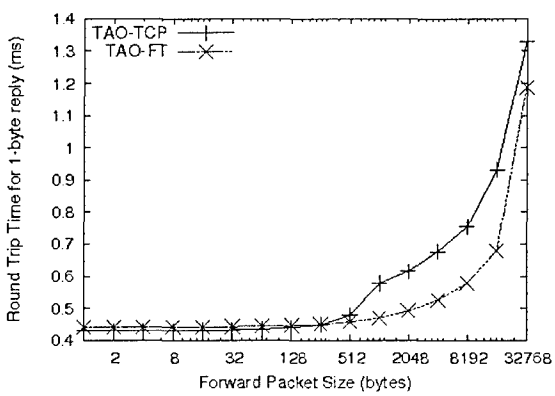

Figure 22: Latency of Transport Mechanisms with 3 Replicas

3.3.5 Overhead of our Fault Tolerance Software

We measured the fault-free overhead of C++/TAO and Java/JacORB versions of our fault-tolerance solution. These tests did not involve the MRLM system, but instead used a simple client-server configuration. Our goal was to compare the latency of using CORBA with raw TCP against the latency of using CORBA with our fault-tolerant middleware (using the Spread GC and MEAD with our duplicate detection and RC enhancement). Since we were not attempting to measure CORBA marshaling cost, we chose the simplest variable-sized data structure that CORBA provides: a sequence of octets. The client sends an octet sequence of a particular size, and the server responds with a single byte. To avoid the inaccuracies associated with comparing timestamps from different machines, the round-trip trip time was measured on the client side.

The results shown in Figure 20 show that our fault tolerance software adds a factor of two to the latency compared to CORBA over TCP. However, if we didn't need replicated servers, then we wouldn't use anything but regular TCP (the whole point of the Replica Communicator). So we also ran the same tests, but with an actively replicated server. Adding a replicated server using our fault-tolerant middleware version was trivial. To implement the replicated server in the TCP version, we constructed a simple sequential invocation scheme where in order to make a single logical call on 2 replicated servers, the client would make an invocation on server instance 1, and then after that call returned the client would make the same invocation on server instance 2. The round trip time for the TCP case is the sum of the round-trip time for both invocations. We implemented a similar setup for a 3-replica configuration. The results are shown in Figure 21 and Figure 22.

In the two replica case, shown in Figure 21, the results show that the fault tolerance software using GC performs nearly as well as TCP, introducing very little extra latency for its total order and consensus capabilities. In the three replica case, shown in Figure 22, the fault tolerance with GC performs better than raw TCP.

3.3.6 Additional ARMS Fault Tolerance Activities

Replicated Security Provisioner

As we prepared for the Gate Test one of the items we hoped to include with the icing results was a replicated Security Provisioner (SP). This component was outside the official scope of GT3 but was something that would run at the global level. It had a number of features and requirements not shared by other MLRM components. One of these was that local Host Security Agents (HSAs) needed to register with the SP via an IP multicast message, which was not supported by our replication framework. Another issue was non-determinism in the SP's behavior.

One of the features we had developed for CCM support was the notion of a clear delineation between start-up time as seen by the application and the start-up time seen by the fault-tolerance portion of the application. By splitting these things apart we enabled CCM bootstrapping to happen before the fault-tolerance infrastructure began to intercept messages and enabled the components to be present before FT made calls on them. This concept is more general than the component use we had previously used it for.

We attempted to make the SP FT using two parallel approaches: making the SP deterministic and dealing with the initial (non-deterministic) multicasting before the FT infrastructure started up. Both of these required software changes and we worked with Scientific Research Corporation (SRC), developer of the SP, to make the changes. We implemented support for an initial period of non-FT execution, where non-determinism could be tolerated. This allowed an SP to register with its local HSA before enabling FT. Unfortunately, it turned out that the effort needed to make the SP deterministic, coupled with schedule pressure, proved to be too large an investment and the integration of a FT SP was not included in the gate test results.

Model-Driven Solutions for Fault Tolerance

During ARMS we defined the concept of a *fault tolerance toolkit*, which was necessitated due to the varying fault tolerance and consistency management requirements of different applications. This required us to envision a fault tolerance solution that can be assembled from smaller building blocks. To this end Vanderbilt University made an initial effort towards a model-driven engineering solution to capture fault tolerance requirements of DRE applications.

The model driven engineering approach is illustrated in Figure 23. These capabilities include defining the failover groups for a group of components that are loosely coupled together, their replication styles, placement constraints, shared risks among components, and others. Interpreters associated with these models synthesize deployment artifacts that are passed on to DAnCE engine so it can deploy and configure the applications with the desired fault tolerance capabilities.

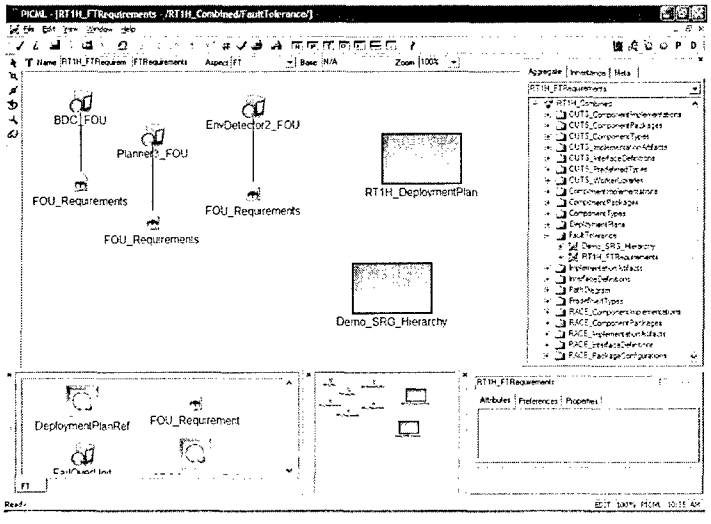

Figure 23: Model Driven Engineering of Fault Tolerance

Since the systems we deal with are dynamic, there is a need for DAnCE to dynamically redeploy and reconfigure application components while keeping the overall mission operational. To that end we are investigating new ideas in redeployment and reconfiguration capabilities within the DAnCE framework. Ultimately our goal is to realize multi-tier fault tolerance capabilities being automatically assembled, configured and deployed using our toolkit as shown in Figure 23.

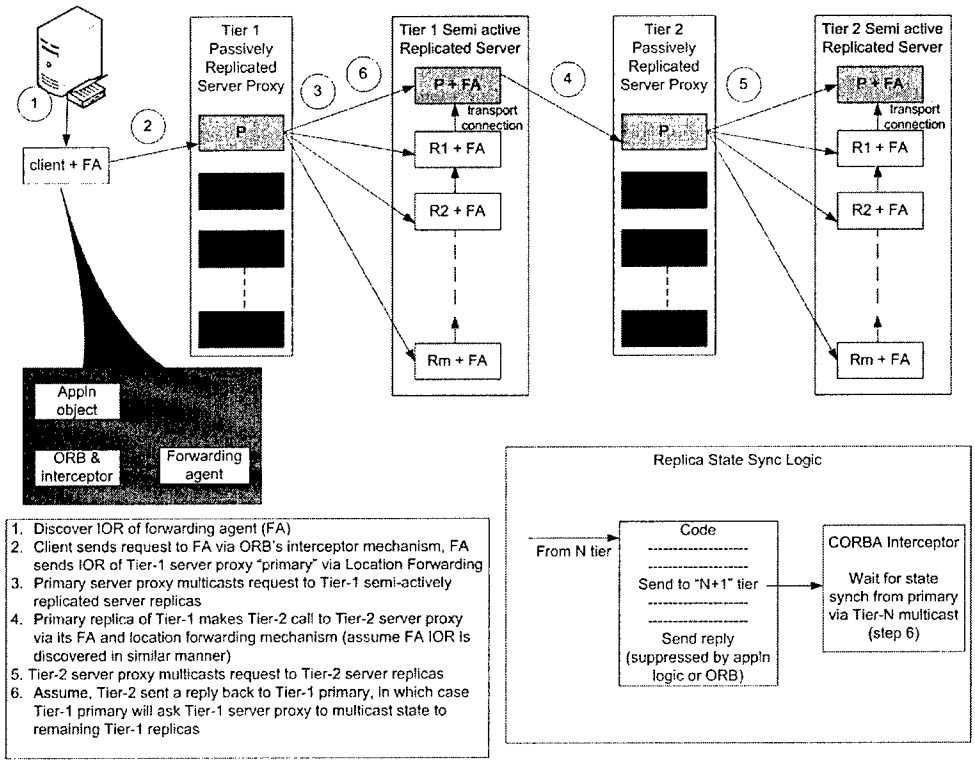

Figure 24: Ideal Model Driven FT Integration

3.3.7 Future Directions and Work in Fault Tolerant Systems

There are a number of areas where future work could provide more efficient, more comprehensive, and more readily usable FT solutions. Replicating clients, supporting periods of non-determinism or non-GC use, and truly dynamic GC systems are all areas where future work could prove fruitful. Also, as we have seen in this and other work even when a solution is available it is not always easy to integrate into a system, even when existing FT solutions are running. In this vein future work on a fault tolerance toolkit could also be very useful.

Passively replicated clients pose difficulties that were largely ignored by previous approaches that focused only on replicated servers. In multi-tiered and peer-to-peer applications, clients (and elements that can be both clients and servers) can be passively replicated, leading to challenges of when state should be gathered and transferred. In Gate Test 3, this problem was exemplified by the Global-RSS passive replicas. For the RSS we added hooks so that the application-level could explicitly trigger state change notifications, which resulted in the primary replica's state being gathered and shared. Another option is to send out state on each client request. A generic framework for adding state to both requests and replies, and a more general state gathering framework would provide support for replication of more application elements.

Currently when a replicated application starts up, there is a time between the application initializing and the time when the FT middleware begins intercepting its traffic. This works well when there are initialization tasks that cannot happen through the FT middleware. There is a more general situation in which an element might need to dynamically (i.e. during normal operation) operate "outside" the FT infrastructure. Currently this is done in an ad-hoc manner with no performance guarantees. Developing a mechanism to more systematically support these different epochs could allow new applications to be replicated for at least some portion of their execution time and also support warm backups if a problem occurs.

Since the conclusion of gate test 3 one of our foundation technologies, the Spread software, has seen a major version change and a number of improvements. One of these improvements is the ability to add daemons during a run without starting all the daemons over again. While this capability is useful, there is additional work in optimizing the daemons that are in use at a given time. Optimizing the message flow taking into account group membership could yield significant improvements in efficiency.

4. Node Failure Detection and Related Transition Activities

With increasing use of COTS hardware, and the associated decrease in cost of large systems, distributed systems with 1000+ nodes are being used to host mission-critical distributed applications. By their nature, mission critical applications often require constant availability. Since no hardware is immune to failures, either from normal wear and tear or from battle damage, these mission critical applications must use some sort of fault-tolerance strategy to provide continuous availability. The Program of Record (PoR) is using a Resource Manager to provide support for activating backups in the case of failures. For the PoR's Resource Manager to provide this functionality, it needs to have accurate and timely notifications of node failures.

The PoR identified the problem of node failure detection (NFD), with the specific scalability and timing requirements for their environment, as a key problem not solved by current COTS or GOTS technology, and requiring additional research and development. This gap in capability, combined with the thrust within ARMS for fault-tolerance techniques provided an excellent transition opportunity.

Lockheed Martin's Advanced Technology Labs produced a partial rapid prototype proof-of-concept implementation of software-based node-failure detection. Starting from that version, BBN developed a complete solution to the PoR requirements including fault-tolerance and scalability, while at the same time ensuring that the solution remained low-overhead. BBN also performed extensive tests of the resulting implementation to demonstrate that it satisfied all of the PoR's requirements. The result of this activity was successful in terms of both advancing the state of the art and in transitioning to the PoR a drop-in technology to fill their node failure-detection gap.

This section describes all the aspects of BBN's Node Failure Detection transition activity. First, we describe the PoR's baseline implementation (B-NFD) and discuss some scalability tests we ran using the baseline. Then, the design and implementation of both Lockheed's initial effort (L-NFD) and BBN's complete solution, including the multi-layer aspects (ML-NFD) are discussed. We then present the results of various experiments using the ML-NFD, and compare them to the results of similar experiments using B-NFD where appropriate. Finally, we provide an account of transition-related interactions with the PoR.

4.1 PoR's NFD requirements

The PoR provided the ARMS project with ambitious requirements for a Node Failure Detector. Individually, some of these requirements might be addressed by COTS or GOTS technology. However, the combination that the PoR required was not satisfied by any existing solutions, and overlapped with the fault tolerance R&D we were doing in the course of carrying out GT3. There were four primary requirements for a technology that filled the NFD role:

Worst-case Detection Time – To support the real-time mission-critical software, and in particular to give the resource management algorithm and mechanisms the greatest amount of time to do their job, the worst-case detection time of a failed node was specified to be under 100 milliseconds.

Scalability – To support failure detection in the context of the PoR's large-scale system the NFD solution needs to scale to 1000 nodes and perhaps more. Combined with the 100ms worst-case detection time requirement, there is potential for significant network traffic that is not directly supporting the mission requirements.

Low Overhead – To enable the real-time mission-critical software to do its job, the NFD will need to run concurrently (i.e., on the same nodes) with this critical software. Therefore, the NFD solution needs to be low overhead in terms of network and CPU utilization. Specifically, no NFD task may take up more than 2% of the CPU time on any single node.

Low False Positive Rate – To prevent wasting resources by unnecessarily triggering backup fail-over, the NFD solution should not generate erroneous failure notifications (false-positives). This is especially important if backup failover of mission critical applications causes non-mission critical applications to be terminated due to lack of resources. The acceptable False Positive Rate was specified as 1 per month.

Fault Tolerance – To remove the existence of a single point of failure, the consumer of failure notifications will be fault-tolerant itself (i.e., replicated). The current requirements indicate that there will be exactly two instances of the failure-notification consumer. Therefore, any failure notification must be delivered to both instances of the consumer. Alternatively, there may be a mechanism whereby two entities (each co-located with one of the consumers) can determine failures independently.

4.2 Design and Implementation of Node Failure Detectors

4.2.1 Program of Record's Baseline Node Failure Detection

4.2.1.1 Architecture

The PoR provided us with a baseline implementation (B-NFD) that uses a TCP-based "pull" model for heartbeats and was designed with some fault tolerance capability. B-NFD consists of client and monitor programs. The monitor program instances run on multiple nodes, to provide fault-tolerance, while client program instances run on all nodes.

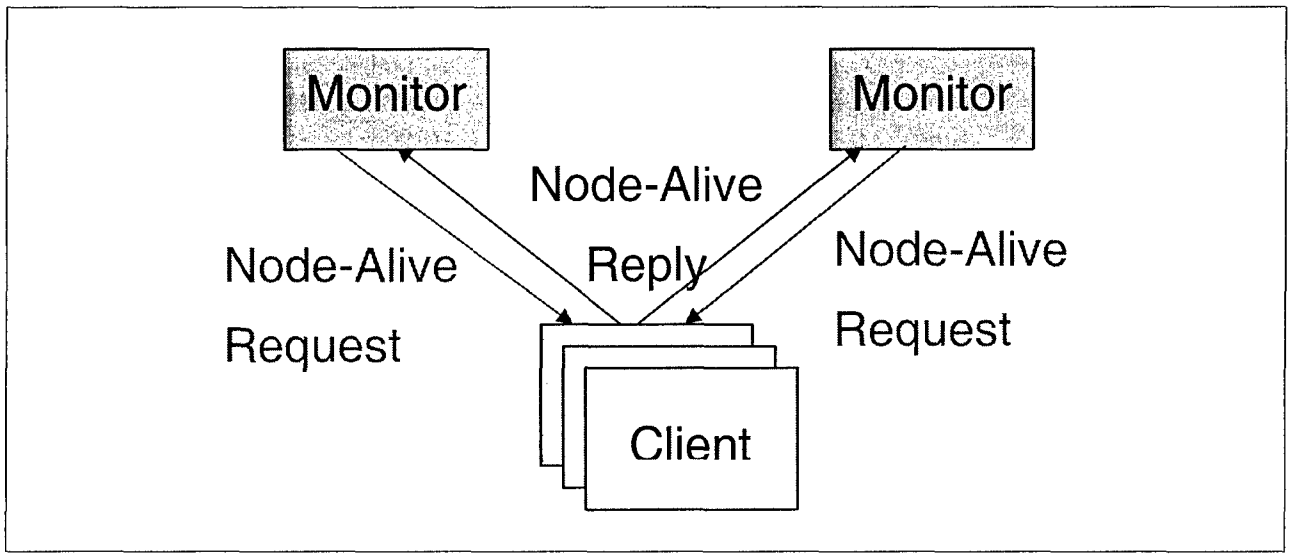

Figure 25: B-NFD Architecture

The interaction between monitor and clients is depicted in Figure 25. For each client, a monitor makes a request for an explicit "node-alive" message, sleeps for a specified amount of time, and then checks to see if a node-alive message was received. It takes note of a missing node-alive message from that particular node and reports a node to be failed if two consecutive node-alive messages are not received.

The client program runs on all physical nodes, and spawns one thread for each monitor. After initially making a connection to the monitors, each thread does a blocking read to wait for requests for node-alive messages from its respective monitor. When a request is received, a node-alive message (reply) is immediately sent to the monitor. Fault tolerance in this design is achieved by having two instances of the monitor act independently, each sending its own requests and getting its own replies. Thus each monitor detects failures independently, and delivers failure notifications to a local consumer instance.

The PoR already believed that the B-NFD was not sufficient to achieve all the requirements given in the previous section. However, to motivate the development of L-NFD, and later ML-NFD, we wanted to prove that B-NFD would not satisfy all the requirements. The version of B-NFD delivered to BBN was configured for 200 ms worst-case detection time. To bring it in line with the requirements from the previous section, we tuned the B-NFD such that, when operating normally, it would detect failures in 100 ms.

We performed some large scale (1000 virtual nodes split evenly between 20 physical hosts) experiments using this tuned B-NFD. The results showed that at large scale, even in the absence of other load on the system, B-NFD used significant resources on the monitor nodes (well beyond the limits set by the PoR requirements), and that the worst-case detection time was higher than the 100 ms requirement. More details on these experiments can be found in Section 4.3.

4.2.2 ARMS Multi-Layer Node Failure Detection

4.2.2.1 Lockheed's NFD

Our collaborator on the ARMS program, Lockheed Martin's Advanced Technology Labs, developed an initial rapid prototype implementation of NFD that was based on a "push" model using UDP. This differed from the "pull" model used in B-NFD since the monitoring nodes did not request node-alive messages. Instead, the per-node sender processes were simply expected to produce node-alive messages every interval (where the value of the interval is configurable, but has a direct impact on the worst-case detection time). By removing the "request" phase, the L-NFD reduces the bandwidth consumed by more than half compared to the B-NFD.

However, Lockheed's implementation followed the same two-tier architecture as the B-NFD, as shown in Figure 25. Some initial scalability tests showed that ~1000 nodes could be handled only if the worst-case detection time was set three times higher than the requirements (i.e., 300 ms). In addition, the CPU usage was 5% on the monitor nodes, which was higher than the stated requirements.

4.2.2.2 BBN's Multi-Layer NFD

There were two areas that BBN believed the L-NFD implementation needed to be improved upon. The first was the combination of satisfying *both* the 100 ms worst-case detection time and the 1000-node scalability requirements. To this end, BBN determined that a multi-layer design, ML-NFD, based on design principles taken from MLRM research results, was more appropriate (see Figure 26) than a single-layer solution. The second area of improvement related to the fault-tolerance requirement. We determined that ML-NFD should be at least as fault tolerant as the B-NFD implementation, and that any additional layers introduced should not lead to a decreased level of fault tolerance. We utilized research results from ARMS fault tolerance R&D to address this issue. Lastly, we implemented a software engineering improvement that turned out to have a significant impact on real-time performance, and hence the false-positive rate: all logging was moved to a dedicated, low-priority thread within each process so that it would not interfere with the primary function of the ML-NFD's processes.

ML-NFD consists of 3 programs: the Node Status Receiver (NSR), Monitor, and Sender. The NSR runs on exactly two nodes (although the latest version can handle an arbitrary number of NSRs). The Monitor program runs on several nodes. Our experiments were run in a configuration that used two Monitors (each on separate nodes) for each 100 Sender instances, to form a cluster as seen in the dotted box in Figure 26. An instance of the Sender runs on every node, and reports to one or more Monitors.

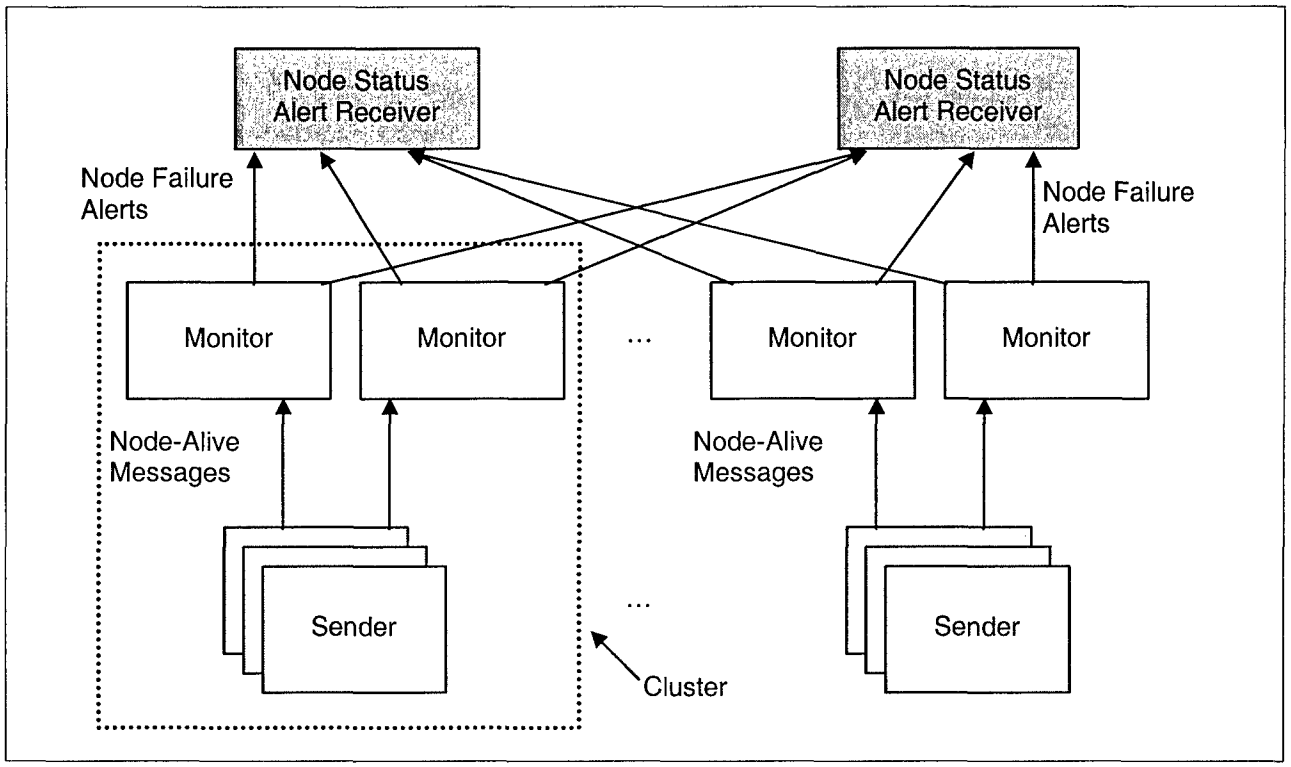

Figure 26: ML-NFD Architecture

Node Status Receiver (NSR)

The NSR is a surrogate for the failure-notification consumer. The NSR is the top-level element of the node failure detection system, primarily responsible for collecting a list of failed or newly "alive" nodes from monitors. Since the design specifies that there may be redundant Monitors for each Sender, the NSR is designed to handle duplicate node status messages appropriately.

Monitor

The Monitor is responsible for processing node-alive messages from the Senders and detecting failed nodes. The node-alive messages are processed in a dedicated thread as they arrive on the UDP port. The detection activity also runs in a dedicated thread, and executes on a periodic basis. The period of the detection activity is configurable, but to provide 100 ms worst-case detection time, we used a value of 40 Hz (every 25 ms). We derived this value by modeling the interactions necessary for notification. Any changes in node status are propagated to the NSR(s) (top-layer).

To determine if a node has failed, the detection activity uses a configurable Detection Threshold (DT). The Monitor loops through a list of all the nodes that it is monitoring, and if the difference between the current time and the time the last node-alive message was received is greater than DT, the node is declared dead. To support the 100ms worst-case detection time requirement, all our experiments used a value of 50ms for DT. Again, this value was derived from our model of the NFD interactions. Note that, by design, DT is not directly related to the rate at which node-alive messages are generated by the Sender. Through experimentation, we determined that the best practice for avoiding a large number of false positives is to have DT at least twice the node-alive interval. This practice allows the system to avoid declaring an erroneous failure in the case of (non-consecutive) single-packet losses.

Sender

The Sender process generates node-alive messages at a configurable rate, and sends them to an arbitrary number of Monitors (determined at process start time). The rate used in all experiments was 45Hz (every 22 ms).

4.2.2.3 Comparison of ML-NFD and B-NFD

The two implementations described above are different ways of addressing the design for a fast, large-scale, and error-free node failure detection. The differences include the model used (Push vs. Pull), the underlying protocol (TCP vs. UDP), and the number of layers. Each of these differences has implications for scalability and the ability to meet or exceed PoR specifications.

Push / Pull

The "Pull" model used in the B-NFD implementation requires that the server issue a request for a node-alive message to each client at regular intervals. Given that the packets in both cases (a request for node-alive, and the node-alive itself) are of equal size (both messages have very small payload size; almost all the bytes "on the wire" are for protocol headers), a "Pull" model uses twice as much network bandwidth as a "Push" model. This also has implications for CPU usage: since the operating system has to do work in the network stack for each packet received, twice as much work is being done in the "Pull" model. The extra CPU usage becomes significant at large scales.

TCP / UDP

There are significant differences in the TCP and UDP protocols that have major design implications. The first difference is that TCP requires acknowledgement (ACK) packets, so even if there were a "Push" implementation that used TCP, there would be extra traffic over a UDP "Push" implementation.

Another difference is TCP's reliability: if message acknowledgements (ACKs) are not received in a timely manner, TCP will retransmit. This has significant implications with respect to predictable timing. The combination of retransmissions with TCP's flow-control mechanisms make the worst-case latency at high load difficult to predict.

Finally, a minor issue is that in general TCP is more expensive in terms of CPU load per packet. Most of this comes from both TCP's larger header and from needing to calculate the checksum on the payload portion of the TCP packet, which UDP does not do. However, in this case the payloads are very small, so this overhead is probably negligible.

Layering

Given unconstrained bandwidth (i.e., the network is grossly over-provisioned), the limiting factor for scalability becomes the number of packets a single node can handle (and at what CPU cost). Using the B-NFD implementation, the nodes that need to handle the most packets are the ones running the server program. Using the ML-NFD implementation, the nodes handling the most traffic are the ones running Monitor programs.

Since nodes in the ML-NFD are divided into clusters that are serviced by independent Monitors, it is possible in the multi layer system to significantly decrease the number of packets that any single node must process.

A multi layer implementation gives enormous flexibility to a system integrator to adapt and optimize the NFD mechanism according to the physical layout of the network. The multi-layer configuration has the capacity to scale much higher than a non-layered configuration. While this design does introduce additional points of failure, we compensate by adding redundancy to each additional point of failure.

4.3 Evaluation of the ML-NFD

4.3.1 Experiment Design

The goal of our experiments was to provide a high level of confidence that the ML-NFD implementation satisfied all the requirements given in Section 4. Our general methodology involved setting up a 1000-node configuration (10 clusters of 100 nodes each), letting it run for several minutes, and then inducing failures. Throughout the experiment, we instrumented the CPU usage of all ML-NFD processes and had instrumentation for detecting false positives.

4.3.1.1 Special Concerns

Since NFD programs will be running on actual physical nodes that are doing real work and generating network traffic, we deployed network load generators on the physical nodes involved in the experiment. There are many ways network load can be introduced in NFD experiments. Two possibilities that represent the "ends of the spectrum" are:

The network load generators are uniformly distributed across all nodes in the experiment. An example of this type of deployment would be if half of the physical machines involved in an experiment run network load sources and the other half run network load sinks.

The network load generator puts load between two physical nodes. There are several variations of this experiment that are unique based on which ML-NFD processes are running on the nodes with artificial network load.

We determined that since the nodes hosting the Monitor processes handle the most ML-NFD network traffic, the worst-case load (i.e., the configuration most likely to cause problems) would be concentrated load between two nodes running Monitors.

The method of inducing faults also warrants special attention. We were interested in verifying correctness of all aspects of ML-NFD under load. Therefore each experiment had three phases, all of which had the load generators continuously executing:

Steady-state, pre-failure: All 1000 nodes active and reporting

Failure of a subset of nodes: half the nodes in a cluster (50 nodes) are failed in the same instant, and timestamps are collected so that we can determine detection time during a post-mortem analysis of the experiment. To gather more data points per experiment, we do this iteratively for each cluster; for each of the 10 100-node clusters, we failed 50 nodes.

Steady-state, post-failure: the remaining 500 nodes (50 per cluster) active and reporting.

4.3.1.2 Experiment Descriptions

The total duration of each experiment was at least 60 minutes. The timeline for each experiment was as follows:

At T = 0 minutes: Experiment Starts, network load is introduced

At T = 31 minutes: Faults are injected causing a subset of nodes to fail.

At T > 60: Experiment is finished.

ML-NFD was subjected to 3 different kinds of network load: high-load (40 Mb/s), low-load (10 Mb/s), and no-load. The B-NFD was only used in a no-load configuration, since the B-NFD used too much bandwidth at the 1000-node scale to add a consistent amount of load. Choice of 40 Mb/s for the high-load and 10 Mb/s for the low-load were influenced by aggregate network load generated by the detectors themselves and the network throughput (100 Mbits) supported by their NIC cards.

4.3.2 Experiment Results

Table 12: No Load Results

Experiment	***Maximum Detection Time at NFA in ms***	***Number of false positives***	***Network Load At Monitor in Mbits. Actual (Expected)***	***CPU load at Monitor in %. Average (Observed Min and Max)***
B-NFD	164	0	13-78 (71.5)	>25* (25,60)
ML-NFD	80	0	2.34 (2.06)	<1 (0,8)

Table 13: Moderate Load Results

Experiment	Maximum Detection Time at NFA in ms	Number of false positives	Network Load At Monitor in Mbits. Actual (Expected)	CPU load at Monitor in %. Average (Observed Min and Max)
B-NFD	N/A	N/A	N/A	N/A
ML-NFD	92**	0	12.14 (11.86)	<1 (0,8)

Table 14: High Load Results

Experiment	Maximum Detection Time at NFA in ms	Number of false positives	Network Load At Monitor in Mbits. Actual (Expected)	CPU load at Monitor in %. Average (Observed Min and Max)
B-NFD	N/A	N/A	N/A	N/A
ML-NFD	90**	0	41.55 (41.27)	<1 (0,8)

*In the low load case the CPU load for the B-NFD was very uneven. With such a high standard deviation the average was not very insightful. However, we know it was greater than 25%, which is much greater than the goal of 2%.

**In the low and high load ML-NFD the difference in detection time would seem to indicate the higher load makes for better detection time. This is not the case and the difference of 2 ms is just "noise" and the numbers together should be taken to show that load does not greatly affect the detection time.

4.3.3 Experiment Analysis

Generally speaking, there are two classes of problems that could arise with either implementation. If there is a problem with the network or a host scheduling problem on the client/sender side, packets are either lost or delayed. The manifestation of this class of problem would be in the form of false-positives. If there is a scheduling issue on the monitor nodes such that their work loops are not running at the intended rate, then the NFD system could exceed worst-case failure detection time requirement.

4.3.3.1 Failure Detection Time

There are several factors that could lead to an out of specification detection time, all of which have the basic property that the monitor work loop is not running at the proper rate.

The first possibility is that CPU load on the system could be preventing the process from running as often as it needs to. This is remedied by using the real-time scheduling class in a stock Linux kernel. We used sched_setscheduler system call (with SCHED_FIFO as one parameter) to get the ML-NFD processes to execute in the real-time scheduling class.

Using relative-time sleep mechanisms can also affect the rate. The standard mechanism used for sleeping (the `nanosleep` system call) is subject to accumulation of error. This is a minor problem, but can cause undesirable side effects such as running at a slower-than-intended rate. The solution is to use an absolute-time based sleep mechanism, such as the `clock_nanosleep` system call.

The final factor is an implementation detail, but an important one. If the monitor has an expensive operation within its failure detection loop, it opens the possibility that some failures may not be detected within the required time. For instance, if each failure event is issued as it's seen (whether via printf, CORBA call to some other component, etc.), and lots of failures happen at once, it's possible that the last nodes to be "detected" in a particular sweep will be beyond the specified worst-case detection time. The way we dealt with this issue in ML-NFD was to queue failure events in the detection loop, and then after the loop is done issue all the failure events for that iteration in one operation.

4.3.3.2 False Positives

There are several scenarios that could cause false positives (reporting that a node is 'dead' when it isn't).

Heartbeat-based solutions are susceptible to network 'hiccups'. In particular, since both implementations are configured such that two missed "node-alive" messages means that a node is dead, a 'hiccup' that caused either two consecutive packets to be dropped or caused any packet to be delayed for a time equal to twice the period at which they are expected would trigger a false positive. The statistical chances of a 'hiccup' (and its nature: dropped vs. delayed packets) vary greatly with the exact network hardware, topology, and workload used.

During over 20 hours of testing with 100 and 1000 node configurations, both under load and unloaded, we observed exactly one false positive for each of the B-NFD and the ML-NFD implementations. We believe that the cause of these two false-positives is the aforementioned 'hiccup'. The root cause of such a hiccup is difficult to determine, especially given the experimental test-bed that we used. After we observed the false positive for the ML-NFD implementation, we added extra instrumentation to help determine the root cause, but never observed another false positive.

Just as a network hiccup could cause a false-positive, a CPU scheduling hiccup on either the client/sender nodes or the monitor nodes could cause delays that lead to false-positives. While CPU scheduling hiccups are somewhat mitigated by using the real-time scheduling class (SCHED_FIFO) for all the ML-NFD programs, later experiments with CPU load indicate that a stock Linux kernel is not sufficient to guarantee consistent scheduling at the granularity that is required for sub-100ms detection times. The PoR was using a real-time version of Linux, so this issue was determined to not be an immediate concern for the PoR.

However, gracefully (and correctly) handling "stock" (i.e., non real-time hardened) versions of Linux at very fine time granularity is a difficult problem, and we believe that its solution would make ML-NFD more generally applicable outside of the PoR context. To this end, we continued work on ML-NFD, and in particular have been implementing and evaluating adaptation strategies that allow ML-NFD to perform acceptably on non-real-time platforms (see Section 4.5).

4.3.3.3 Resource Usage – Network

The difference in network usage is significant. Analysis of theoretical usage shows almost a factor of two difference between B-NFD and ML-NFD. (Note that packet sizes include all headers: TCP/UDP/IP/Ethernet)

B-NFD: Average packet size: 71 Bytes (as measured by Ethereal)

71 Bytes * 33 Hz * 2 (for each server node) * 2 (one request packet, one reply packet) * 8 (bits / byte) = 0.072 Mb/s per node.

ML-NFD: Packet size: 60 Bytes (as measured by Ethereal)

60 Bytes * 45 Hz * 2 (for redundant Monitor node) * 8 (bits / byte) = 0.041 Mb/s per node

At 1000 nodes, with the expected level of fault-tolerance (1 redundant server for B-NFD, 1-redundant Monitor per cluster and 1 redundant NSR for the ML-NFD), the expected aggregate network load (i.e., over all the networks in the system) generated by B-NFD is 71.5 Mb/s and by ML-NFD is 41.2 Mb/s.

The worst-case load at any single node shows an even greater disparity, given that ML-NFD has only 100 nodes reporting to any given Monitor:

B-NFD server node: 71 Bytes * 33 Hz * 1000 nodes * 2 (request/reply) * 8 = 35.75 Mb/s.

ML-NFD Monitor node: 60 Bytes * 45 Hz * 100 nodes * 8 = 2.06 Mb/s.

What is particularly interesting regarding the network usage is the difference between the expected load for the B-NFD version and the actual observed load at the client and monitor nodes. When running 50 clients on one physical node, the expected network load at that physical node is 3.58 Mbits per second. The observed load was a very consistent 4.2 Mbits per second. An explanation for this is that TCP is re-transmitting some packets. At the monitor node, the B-NFD load varied widely between 13 Mbits and 78 MBytes. This bursty behavior seems to reinforce the theory that TCP retransmissions are happening.

For ML-NFD, actual network usage at the Monitor and Sender nodes is very close to expected network load. For all configurations that we tested, actual network usage exceeded expected network usage by approximately 0.28 Mbits.

4.3.3.4 Resource Usage – CPU

The ML-NFD implementation is efficient in terms of CPU usage on the sender and monitor nodes. B-NFD implementation is only efficient on client nodes. The dominant factor in CPU usage for both implementations is directly proportional to the network load at that host. Since the operating system must do work for each packet, the CPU usage is directly proportional to the number of packets sent and received. A secondary issue is that TCP packets are generally more CPU intensive for the operating system than UDP packets, given the extra features (and complexity) of TCP.

For 1000-node experiments, the observed utilizations at the client for B-NFD and sender for ML-NFD nodes were negligible, but utilization at all the monitor nodes in ML-NFD (each monitoring 100 senders) was less than 1% while the monitor nodes in B-NFD had 25%-60% CPU utilization. It is safe to assume that this wild variation is correlated with the observed network-load variations mentioned in the previous section.

4.3.3.5 Scalability

The factors that limit scalability of all NFD implementations include total network capacity and the per-node CPU power and network bandwidth. Our experiences with both the B-NFD and L-NFD implementations indicate that the latter (per-node CPU and network) restriction was the limiting factor in all the configurations we used. Experiments showed that both these implementations had problems with large scale (1000-node) deployments.

The ML-NFD design addresses the per-node limiting factor by putting a hard limit on the amount of work (both pure CPU and handling network traffic) that needs to be done by any single node regardless of the total number of nodes.

The size of the clusters determines how much CPU and network traffic will be required at the Monitor nodes. The key to ML-NFD's scalability is that there is virtually no additional periodic (steady-state) overhead on existing nodes when new clusters are added. This means that the limiting factor in ML-NFD scalability is the total network capacity.

4.4 Transition of ML-NFD to the PoR

The ML-NFD was successfully transitioned to the PoR. In the course of this transition effort, we delivered several versions of software, along with documentation and experimentation results, to PoR personnel and worked with the PoR to help them evaluate it and integrate it. The following timeline shows the important milestones of the ML-NFD transition effort.

Jan 18, 2006: BBN personnel visit PoR personnel at the PoR site in Portsmouth, RI to discuss transition possibilities of the aspects of our Gate Test 3 and fault-tolerance work that fit the PoR's identified needs. PoR gave us a copy of B-NFD for reference.

Mar 02, 2006: Lockheed publishes to PoR L-NFD (ARMS-NFD version 1) and results for 1000-node experiments using 300ms worst-case detection time.

March 24, 2006: BBN publishes to PoR the first version of the ML-NFD implementation (ARMS-NFD version 2), along with preliminary experimental results.

May 11, 2006: BBN personnel visit the PoR site to discuss ML-NFD integration and the "network load" suite of experiments.

June 12, 2006: BBN publishes a new version of ML-NFD to PoR (ARMS-NFD version 3). This version included various instrumentation items that were added to conduct our experiments. During discussions with the PoR, they indicated interest in these additional instrumentation items, so we packaged a new version and delivered it.

August 03, 2006: BBN publishes a new version of ML-NFD to PoR (ARMS-NFD version 4). This version included transition of the primary programming language from C to C++, improved logging that would not interfere with real-time deadlines, and the ability to turn instrumentation on and off at run-time.

August 04, 2006: Telecon between BBN and PoR to discuss ARMS-NFD version 4.

August 18, 2006: BBN publishes a minor improvement of ML-NFD to PoR (ARMS-NFD version 4.1). This version included a network interface to turn the instrumentation on and off remotely and used implicit node identification based on IP address (previous versions used an abstract *Node ID* that in practice would have needed to be mapped into an IP address by an external entity).

September 29, 2006: Telecon between BBN and PoR to discuss PoR integration activities.

4.5 Adaptive ML-NFD

The PoR used a real-time operating system (RTOS) so that hard guarantees about CPU scheduling could be made. We believe that ML-NFD would be more generally applicable if there was not a requirement that an RTOS be used. By allowing ML-NFD to be used on a greater range of commodity hardware and software, its value is multiplied. The greatest challenge when using a non-RTOS is that CPU scheduling can be more erratic.

To get a feel for what would happen if ML-NFD were deployed in a non-RTOS environment, we performed several small-scale (approximately 10 nodes), long running (1-2 weeks) experiments on workstations running several varieties of non-real-time Linux. These workstations were in use by developers doing normal work throughout the experiments.

This "normal load" induced a surprising number of false positives; some nodes averaged over 200 false positives per day. There were many CPU scheduling hiccups where the periodic portions of ML-NFD processes (Sender and Monitor) did not get to execute for over 100ms (which guarantees a false positive). Clearly configuring ML-NFD for 100ms worst-case detection time on non-RTOS systems would not work in all cases. However, some nodes had relatively few false positives (averaged 1 per day).

We believed that if the requirement for 100ms worst-case detection time was relaxed, it should be possible to get the false positive rate down to a very low level (average 1 per month). The question then became: how *much* should we relax the requirement by, and should it be done for all nodes, or on an individual basis?

The remainder of this section describes some of the adaptation strategies we have developed and tested for the ML-NFD implementation.

4.5.1 Per-monitor, cluster-based detection-threshold adjustment

Our initial adaptation technique was based on the presence of groups of failures. Initial tests using a driver program to simulate failures showed that this technique worked as designed. However, in extended tests without the simulated failures it was not as effective at avoiding false positives.

4.5.2 Per-monitor scheduling compensation

Analyzing the code and logs from extended runs, we identified a specific case that would cause all of the nodes being monitored (by a given Monitor instance) to be (false positively) declared as failed. We determined the cause to be a scheduling hiccup in the Monitor node.

The adaptation strategy that we developed was to have the monitor's periodic failure-detection thread notice when it doesn't get to execute on schedule, and to ignore reported failures for that cycle. The impact is slightly increased worst-case detection times. This seems to be an acceptable tradeoff, since the scheduling hiccups (of duration h) are often much larger than the intended period of the failure-detection thread (p, which by default is 25 ms). Without this adaptation, the worst-case detection time is determined by h. With this adaptation, the worst case detection time is determined by $h+p$.

Extended tests showed that this technique prevented nearly all the "clusters" of false positives. This accounted for approximately half the false positives declared during the tests.

4.5.3 Per-monitor, per-node detection-threshold adjustment

While the Per-Monitor Scheduling Compensation deals with groups of failures, we also needed a way to deal with individual nodes that are unable to schedule their Sender process consistently at the initial rate, and thus lead to a high false-positive rate.

We have implemented an adaptation that allows the failure-detection threshold to be adjusted (strictly increasing) on a per-node basis. This allows the NFD software to "discover" appropriate thresholds for each individual node to avoid repeated false-positives. At the same time, it allows us to keep tight bounds on worst-case detection time for nodes that are behaving more like real-time operating systems (i.e., no scheduling hiccups). The threshold adjustments are propagated to the NSR (top-tier) nodes, so that resource-management software that consumes node failure events will be able to take worst-case detection time differences in nodes into account.

We are still evaluating this adaptation in combination with the Per-monitor Scheduling Compensation technique.

4.6 NFD Conclusions

The PoR came to ARMS with ambitious requirements to detect failures in a timely, accurate, and efficient, and fault-tolerant manner. Building upon a proof-of-concept implementation and applying ARMS multi-layered design and fault tolerance R&D results, we were able to increase the timeliness of the system by making the sending and receiving of heartbeats more efficient and less susceptible to load on the host system, both on a per-host basis as well as in the overall system. This leaves more time for the application-level work of the system to complete while also limiting the effect of the network performance on our detection capability. We improved accuracy by carefully engineering the timers used by the system and by allowing detections to be delivered under the imposed time limit. Lowering both the CPU and network utilization increased the efficiency of the system, while having redundant monitors increased the systems fault-tolerance. All of these changes were thoroughly tested in a number of scenarios before being transitioned back to the PoR for inclusion in the operational system under development. In addition, developing self-adaptation capabilities for false positive management serves as a significant step forward in the state of the art for node failure detection.

5. Dynamic Resource Management

In this section we describe BBN activities to develop dynamic resource management (DRM) methods for the ARMS MLRM system. A goal of the ARMS project is to design a runtime computational resource allocation engine provides appropriate and sufficient system resources such as CPUs, memory, and bandwidth to be allocated for application strings to accomplish user specified tasks across a shared, common computational infrastructure. If there are insufficient resources to accomplish all desired tasks, resources should be allocated to applications strings so the value to the warfighter of the computations performed are maximized. In addition, the system's resources should be allocated as swiftly as possible and with enough additional resources in reserve if possible in order to best cope with unforeseen contingencies that might otherwise require the reallocation of scarce resources.

At a high level the resource management goals in ARMS can be summarized as follows:

Maximize war fighting capability – Maximize string uptime and meet end-to-end real-time deadlines

Deployment based on relative importance – deploy important application strings ahead of other strings

Best utilize available resources – deploy as many application strings as possible with best possible resource utilization efficiency

In the ARMS/MLRM system, we established that system behavior can be decomposed into various missions. In the POR context, the system might have a radar tracking mission, a target assignment mission and navigation mission among others. Some of these missions would be relatively more important than others and their relative importance rankings may change over time and situation. For instance, when approaching port during stormy weather, navigation may be more important than target assignment, but conversely, during battle conditions, target assignment may be more important than navigation.

Every mission in the system can generally be decomposed into possibly repeated sub-missions called strings. For example, a radar-tracking mission might have multiple strings that correspond to the tracking of various targets. As with the missions, some strings might be of more importance than others. To again revisit the radar-tracking mission, it will likely be more important during some operating conditions to track enemy warplanes more accurately than friendly aircraft. The decomposition of global system behavior into missions and missions into strings is depicted visually in Figure 27.

As part of our dynamic resource management R&D activities, we developed a set of utility functions as real-time measures of system performance. These utility measures were intended to be high fidelity versions of the two warfighter value metrics in the GT4 CONOPS document. We defined one utility function, called the *application utility,* to be both a real-time and an evaluative measure of how well the system accomplishes the tasks given to it be the user. This measure could be used by a DRM system for online performance evaluation when making resource management decisions. We defined another utility function, call the *resource utility*, to be a measure of how efficiently DRM actually uses the resources.

Because of the natural system-mission-string decomposition of the ARMS MLRM, we designed separate utility functions for each layer of the hierarchy where higher layer utility functions are composed from lower layer utility functions. This utility function approach to integrated dynamic resource management also provides a more rigorous and potentially more easily understood approach than functionally similar ad hoc methods.

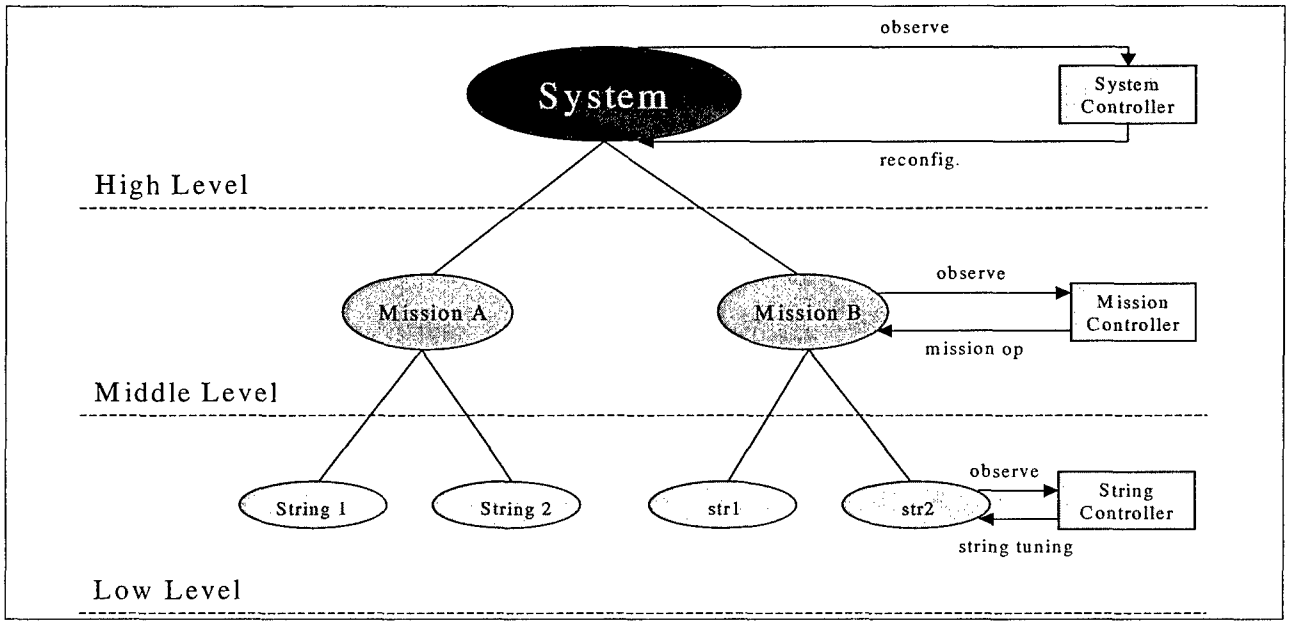

Figure 27 Three-tiered control system hierarchy

We used our utility functions to guide our design of a hierarchical resource management system based on a string-mission-system decomposition of system behavior. Our hierarchical control system was designed to use multiple local resource allocation algorithms to tune local resource management behaviors at the string, mission and multi-mission levels. We investigated multiple resource management algorithms that could be used at each level of the DRM hierarchy.

As we were designing our control system, we developed Matlab/Simulink simulations of the ARMS system to examine the benefits of the various resource control algorithms in the context of the warfighter measures from the GT4 CONOPS. We used these simulations to select a set of algorithms for the control hierarchy that we implemented in the GT4 testbed and to contribute to the GT4 icing efforts. Using our algorithms, we were able to achieve an order of magnitude improvement in DRM performance as measured by the GT4 warfighter value metrics.

5.1 Utility Functions for DRM Performance Measurement

In order to guide resource management strategies, we defined real-time and evaluative measures of ARMS DRM performance that are more expressive and take into account more variables than the baseline warfighter value metrics presented in the GT4 CONOPS. These observable utility measures for the ARMS system that can be used to objectively compare two systems' performance operating under different resource management strategies from a user's perspective with a higher fidelity than the GT4 CONOPS warfighter metrics. Our goal was for the utility measures to be used as a guide for the DRM system to decide when and how DRM behavior would need to be adjusted when the system's real-time performance degrades. The ultimate goal of utility function development was to develop a system performance measure that should be maximized by the DRM during system operation.

We defined utility functions to measure both the *Application Utility* function and the *Resource Utility*. The application utility function is used to evaluate the user-perceived value delivered by the system. The resource utility function is defined to evaluate how efficiently and quickly system resources are allocated and utilized.

We present some definitions before the introduction of the application utility functions and resource utility functions.

- **Application** – An application is an instance of code running on a node.
- **Task** – A task or end-to-end task is a logical sequence of applications that perform countable units of effective work subject to warfighting QoS requirements.
- **Job** – A job is defined as each invocation of a task.
- **Application String** – An application string often has multiple tasks that together help to meet the end user's requirements (which can either be hard constraints or soft operational ranges). End-to-end requirements and utilities are generally associated with specific end-to-end tasks in an application string. One task cannot belong to more than one string.
- **Mission** – A high-level, possibly repeated, objective that is composed of strings.

5.1.1 Application Utility

The application utility in a system is computed by taking the weighted sum of the Application Utilities of the system's missions, and a mission's application utility is computed by summing the application utility of the mission's strings. We first show a method to compute the application utility of a string before defining the application utility of a mission and the entire system. Factors contributing to application utility include:

Availability – Is the application string or system up? Services can only be provided to the end-user when the string or system is operating and free of failures.

Timeliness – Are the end-to-end deadlines met? Missed deadlines can have negative, sometimes catastrophic, impact on the system.

Quality – How accurate or complete is the information or data delivered by an application string? Data compression, filtering, data transformation and incomplete transmission of information, as side-effects of operating in a network-centric environment, can all affect the quality of information delivered by the application string in a potentially adverse manner.

Throughput – For repeated jobs, this is a measure of how well an application string is able to achieve its desired throughput during a given time interval.

We discuss how we perceived the above factors should contribute to the application utility of a string. We then show how we incorporated these factors into a comprehensive utility function for the evaluation of application utility. This utility function is only one of several valid and plausible functions for the assessment of application utility.

5.1.1.1 Timeliness

End-to-end tasks typically have real-time deadlines associated with them. If each task can have multiple jobs (periodic, aperiodic or sporadic), and deadlines are either met or missed for each job in a task. Missed deadlines could potentially diminish the warfighting capability of an application string (soft or firm real-time) or even result in failure of the mission (hard or firm real-time). There are several possible scenarios for assessing the utility of a job's timeliness according to whether a given deadline is met and whether the deadline is hard, firm or soft real-time:

If the deadline is met, then some utility or value y ($0 \leq y \leq 1$) can be given for that job.

If a hard real-time deadline is missed, there can be a penalty p ($-1 \leq p \leq 0$) associated with missing the deadline.

If a firm real-time deadline is missed, the job has a given value y ($0 \leq y \leq 1$) if the total number of missed deadlines in a constant time window τ is below a given maximum. However, if the total number of missed deadlines is over the given maximum, then the firm real-time deadlines become hard real-time deadlines. Therefore, after the maximum number of allowed missed deadlines has been reached in a specific time window, the penalty p ($-1 \leq p \leq 0$) is associated with missing all future firm real-time deadlines in the time window. In this situation the number of missed firm real-time deadlines needs to be tracked using a fix-sized sliding window. For notational simplicity, define $thr(t,\tau)$ to be a Boolean function that returns true if at time t the number of firm real-time deadlines missed in the window τ has passed the threshold and false otherwise.

If a soft real-time deadline is missed, then it is possible for there to be a partial utility associated with completing the job after the deadline. This partial value could depend on the amount of time after the deadline that the job is completed so that the longer the time, the smaller the utility. There are different approaches to capture this concept of diminishing value: one possible approach is to use the ratio of a job's deadline d to its actual execution completion time ec *(where* $ec > d$*),* d/ec.

Therefore, based on the above discussion, a job will get either a non-zero value γ or a non-zero penalty ρ, but not both. (That is, either γ or ρ will always be zero, but not both.) This is due to the intuition that one cannot be rewarded and penalized at the same time. Suppose the real-time task types are mapped to the following R values:

$R = 1$ hard real-time.

$R = 2$ firm real-time.

$R = 3$ soft real-time.

The following functions can then capture the value γ (Equation 1) or the penalty ρ (Equation 2) under the scenarios discussed above.

$$\gamma = f(ec,d,R,t) = \begin{cases} y & ec \le d \wedge (R=1 \vee R=2 \vee R=3) \\ y & ec > d \wedge R=2 \wedge \neg thr(t,\tau) \\ d/ec & ec > d \wedge R=3 \\ 0 & otherwise \end{cases} \quad (1)$$

$$\rho = f_\rho(ec,d,R,t,\tau) = \begin{cases} p & ec > d \wedge R=1 \\ p & ec > d \wedge R=2 \wedge thr(t,\tau) \\ 0 & otherwise \end{cases} \quad (2)$$

5.1.1.2 Quality

In addition to meeting real-time deadlines, the quality of the delivered information due to a job is also of importance to a warfighter. When data is moved from producer to end consumer and processed by a number of applications along the way, the delivery of the data may be delayed, part of the data may be lost, or the data reaching the consumer may become transformed. For example, data compression and filtering techniques used to reduce bandwidth usage could irrecoverably degrade the quality of the data transmitted between the data producer and data consumer. Let q ($0 \le q \le 1$) be a measure of the relative change in quality of a set of data after it has been processed by a job. If $q=1$, then the data processing performed has caused no data loss. If $q=0$, then there has been a complete data loss caused by application processing. The value q can be measured, reasonably estimated or experimentally determined. Furthermore, a value for q can be assigned in the application mode where data processing occurs.

For an example of how data quality can affect the utility of a resource allocation, consider the example illustrated in Figure 28. In delivering air search radar information, various forms of data could be suitable depending on the resource availability and mission requirements. To reduce the amount of data transmitted, raw sensor data can be converted to a two-dimensional image. This conversion/compression is often lossy.

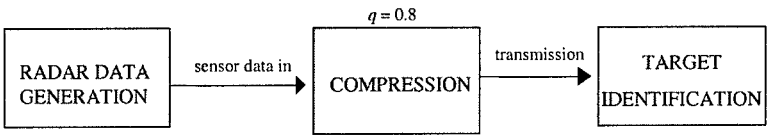

Figure 28: An example of processing techniques that can impact the quality of radar sensor data

In this example, the radar data needs to be compressed and sent over a communication link to a target identification application. Due to limitations in the allocation of computation power and bandwidth, the radar data is compressed in a lossy manner that results in 20% data loss. Therefore, this job has a data quality factor of q = 0.8.

Note that there may be multiple methods for measuring a change in data quality. One approach would be to include an information-theoretic entropy measure of data corruption.

5.1.1.3 Availability

Unlike the timeliness and quality factors discussed above, availability is measured on a per string basis and not on a per job basis. For simplicity, it is considered that a string is down when any of its individual applications is down due to failure. A string adds no value during down time. Let T_{up} be the time that a string is up during a time T that is the time window for the evaluation of availability[2]. The availability factor a for the string can be expressed as the ratio:

$$a = \frac{T_{up}}{T} \qquad (3)$$

[2] This time frame is generally different from the sliding time window for the firm real-time tasks.

5.1.1.4 Throughput

Like availability, throughput is measured on a per string basis. Generally it is desired for a string with repeated jobs to successfully process as many jobs as possible over a given time interval without missing any deadlines. *Throughput* is a measure of the rate at which a string processes repeated jobs over the time T. Naturally, this measure is relevant only to jobs that need to be repeated regularly.

Generally, the higher the throughput of a string, the better, but like timeliness, there can be several cases for how the relative desirability of a given throughput depends on the rate of job processing. For instance, if given a maximum desired processing rate α_{max}, the processing rate $\alpha \leq \alpha_{max}$ could have a throughput desirability factor of α/α_{max}. This case is called *relative throughput desirability* because the desirability of a throughput is relative to a maximum throughput. However, in the case of *threshold throughput desirability*, if a processing rate α_1 is strictly greater than the processing rate α_2, but both α_1 and α_2 are greater than a threshold α_{th}, then α_1 and α_2 have the same throughput desirability factor of 1. Furthermore, for a throughput $\alpha_3 < \alpha_{th}$ that does not exceed the desired threshold, then the throughput desirability factor could be 0. The intuition behind threshold throughput desirability is that the threshold α_{th} represents the minimum throughput a string needs to have in order to accomplish its task.

A composite throughput desirability model that combines relative throughput desirability and threshold relative throughput desirability can also be defined where if a processing rate α_1 is strictly greater than the processing rate α_2, but both α_1 and α_2 are greater than a threshold α_{th}, then α_1 and α_2 have throughput desirability factors of α_1/α_{max} and α_2/α_{max} respectively. Furthermore, for a throughput $\alpha_3 < \alpha_{th}$ that does not exceed the desired threshold, then the throughput desirability factor is 0.

Although only cases of relative throughput desirability, threshold throughput desirability and composite relative threshold throughput desirability are considered here, it is entirely conceivable that other methods exist to measure the relative desirability of the throughput of a string.

Suppose the throughput desirability types are mapped to the following Γ values:

$\Gamma = 1$ relative throughput.

$\Gamma = 2$ threshold throughput.

$\Gamma = 3$ composite relative threshold throughput.

The following function captures the throughput desirability factor under the scenarios discussed above.

$$Th = f(\alpha, \alpha_{\max}, \alpha_{th}, \Gamma) = \begin{Bmatrix} \alpha/\alpha_{\max} & \Gamma = 1 \\ 1 & \alpha \geq \alpha_{th} \wedge \Gamma = 2 \\ \alpha/\alpha_{\max} & \alpha \geq \alpha_{th} \wedge \Gamma = 3 \\ 0 & otherwise \end{Bmatrix} \quad (4)$$

If the string throughput α is controllable, then by adjusting α, the string may be tuned to trade off decreased string throughput for a higher timeliness factor in the case of resource contention. In this case, the throughput of a system could be adjusted to optimize the utility of a string.

For an example of a system with a relative throughput measure, consider a string deployed for a radar tracking system. The throughput of the radar processing string is the rate at which successive tracks of the same target can be successfully resolved. Naturally, it is preferable for the string in this radar-processing example to be able to process all given tracks at the same rate at which the tracking data is generated. The higher the throughput for the radar processing system on any given interval, the better. However, due to unexpected limitations in computation power or a surge in trackable objects, the string may not be able to process tracks as fast as desired while avoiding missed deadlines. By decreasing the processing rate, we could process the remaining tracks so that their deadlines are met. If Θ_{max} is the maximum desired throughput for the tracking system and $\Theta \leq \Theta_{max}$ is the actual throughput, then Θ/Θ_{max} is the throughput desirability for this system.

As would be expected, not all strings have a throughput measure as throughput has no meaning for strings with jobs that are not repeated. In this case, if a string is not repeated, then the throughput desirability can always be assigned.

5.1.1.5 Defining Application Utility

To incorporate availability, quality, throughput and timeliness factors into an integrated application utility function, we first designed strings functions for jobs and tasks that take into account timeliness and quality. Then, a string level application utility function was developed that accounts for availability, timeliness, quality and throughput. This string-level application utility can then be generalized for both single missions and holistic multiple-mission systems.

For an individual string, if a job is penalized because it missed its deadline (hard or firm), then the quality factor is no longer important and the penalty ρ should be the utility (albeit negative utility) of the job. However, if a job is rewarded with a value γ, then one would compute the utility of a job by taking the product of the value and the quality factor (1 is considered to be the maximum utility of a job). Therefore, the following function can be used to capture the utility of the i^{th} job u_i in a task with respect to timeliness (value or penalty), and quality.

$$u_i = \rho_i + \gamma_i q_i$$

Recall from the discussion of the timeliness factor in Section 5.1.1.1 that either γ_i or ρ_i will be zero, but not both. Therefore, if ρ_i is zero, then $u_i = \gamma_i q_i$, and if γ_i is zero, then $u_i = \rho_i$.

In a time frame of length T, two tasks could execute a different number of jobs. Assuming all the jobs in both tasks met their deadlines and had perfect information quality, then one task would have a much higher utility simply because it has more jobs (or execute at a higher frequency). Therefore, it is desired to measure the utility of a task by measuring its average utility per job. If P is the total number of jobs for the task in the time frame T for evaluation, one could compute the average utility u_j for a task as:

$$u_j = \frac{1}{P}\sum_{i=1}^{P} u_i$$

An application string may have multiple tasks and the utility of a string could be measured by the average utility of the tasks in the string. For a string S_k, let O_k denote the number of end-to-end tasks in S_k over some sample period. Then, the application utility of the string S_k without taking into account the string downtime and throughput is:

$$UA_k = \frac{1}{O_k}\sum_{j=1}^{O_k} u_j \qquad (5)$$

where u_j is the utility for the *jth* task in the string.

The utility calculated in Equation 5 is over time T under the assumption that the string is up during the entire evaluation and without taking into consideration the string's throughput. Therefore, the application utility of the string S_k taking into account the string downtime and throughput is:

$$UA_k = \frac{aTh}{O_k}\sum_{j=1}^{O_k} u_j \qquad (6)$$

wherein Equation 6, a is the availability of the string S_k calculated using Equation 3 and Th is the throughput of the string calculated using Equation 4. The utility calculated using Equation 6 is essentially the string's effective uptime (the time a string is up, consuming resources and effectively performing tasks that contribute to the warfighting capability), which is a good approximation of the string's actual warfighting contribution.

It is now shown how a string's application utility contributes to the application utility of its mission and hence the application utility of the entire system. Assume there is a set of *str(j)* application strings associated with the mission j. Each string S_i has a relative value w_i associated with it which indicates the importance of each application string to the mission relative to all other application strings and is part of the domain meta-information associated with the mission. For example, a string with an importance value of 0.5 is considered five times more important than one with an importance of 0.1. Importance values selected using different scales should be normalized to provide a consistent view. Therefore, the mission level application utility UA (j) for mission j can be defined as the sum of the Application Utilities of mission j's strings weighted by their importance values as shown in Equation 7:

$$UA(j) = \sum_{i=1}^{str(j)} w_i UA_i \qquad (7)$$

wherein UA_i is the application utility for string i in mission j and it can be computed using Equation 6. Consequently, if there are m missions, then the global application utility is defined as the weighted sum of the component mission's Application Utilities where the weight $w(j)$ is a measure of the relative importance of the mission j:

$$UA = \sum_{j=1}^{m} w(j)UA(j) \qquad (8)$$

5.1.2 Resource Utility

Although the application utility function assesses the ability of strings to accomplish user-specified tasks under particular system resource management strategies, they do not assess how efficiently and quickly system resources are allocated and utilized. Therefore, we defined the resource utility function to evaluate this aspect of an allocation strategy and algorithm. The resource utility function presented is based on the factors of resource slack and speed:

Resource slack – What portion of resources remain available to accommodate further deployments or unforeseen contingencies?

Speed – How fast can an algorithm perform an allocation or reconfiguration?

5.1.2.1 Resource Slack

Resource slack is the measure of the percentage of free resources in the system and it is one way to measure the quality of the allocation produced by an algorithm. It is significant in two ways:

It is an indication of the efficiency of an allocation algorithm. A resource allocation with a resource slack of 0.12 (12% resources free) is better than one with a resource slack of 0.10 (10% resources free) because it is more efficient.

The resource slack also provides a measure of the system's ability to handle additional loads and fault situations without resorting to reconfiguration. Free resources provide a buffer that can be devoted to unpredicted loads and faults without the (more costly) reallocation of resources already devoted to other strings in the system.

There are different approaches to capture the resource slack factor and the specific approach taken is largely dependent on the end requirements of the resource allocation. For example, if a generally low utilization for all resources is important, then the resource slack factor s due to an algorithm can be expressed as a function of the average utilization μ of all resources combined (CPU, bandwidth etc) [3]:

$$s = f_s(\mu)$$

In the context of the work discussed in this paper, the resource slack s is normalized without loss of generality to be a real value between 0 and 1 ($0 \leq s \leq 1$) for mathematical simplicity.

In order to guarantee that low resource utilization has a higher utility than high resource utilization, $f_s(\mu)$ will be strictly decreasing or at least decreasing with respect to μ.. If $f_s(\mu)$ is strictly decreasing with respect to μ and there are two utilizations μ_1, μ_2 such that $\mu_1 \leq \mu_2$, then $f_s(\mu_1) \geq f_s(\mu_2)$.

Similarly, in order to guarantee that conserving resources at high resource utilizations is relatively more valuable than conserving a comparable amount of resources at low resource utilizations, $\frac{d}{d\mu} f_s(\mu)$ should generally be strictly increasing or at least increasing with respect to μ. To demonstrate this, let there be two utilizations μ_1, μ_2 such that $\mu_1 \leq \mu_2$ and let there be a value ε that represents a possible amount of decrease in these resource utilizations. Then, $f_s(\mu_1)$- $f_s(\mu_1-\varepsilon)$ is the relative utility of removing an ε amount of resources from the utilization μ_1 and $f_s(\mu_2)$- $f_s(\mu_2-\varepsilon)$ is the relative utility of removing an ε amount of resources from the utilization μ_2. Because $\frac{d}{d\mu} f_s(\mu)$ is strictly decreasing, then $f_s(\mu_1)$- $f_s(\mu_1-\varepsilon) \leq f_s(\mu_2)$- $f_s(\mu_2-\varepsilon)$ which implies that the utility of removing the ε amount of resources at utilization μ_2 is greater the utility of removing the ε amount of resources at utilization μ_1.

[3] The allocation algorithm can be designed to run on a dedicated node and consequently $f_s(\mu)$ does not need to take into account the overhead associated with running the allocation algorithm.

The relative curvature of $f_s(\mu)$ with respect to μ can be tuned to properly avoid higher levels of resource utilization based on contextual knowledge of the system. However, a simple approach for implementing $f_s(\mu)$ used in this version of the paper is a linear function of the average utilization of all resources (CPU, bandwidth etc), μ, as shown below:

$$s = f_s(\mu) = 1 - \mu$$

Another possible approach is to use the peak utilization among all resources as the measure of resource slack as is commonly done in the field of power generation. Note that in the case that not all resources are valued equally, $f_s(\mu)$ could be generalized to be a function with a multi-dimensional input that more explicitly conveys the relative utilizations of the resources. One example could be that the shared resources in the system are given different relative importance values, and the benefit of achieving a higher resource slack is ranked by each resource's importance, i.e., maximizing the slack for the most important resource is more valuable than for the second most important resource etc. In this case the input to the resource slack function could be a vector of resource utilizations.

5.1.2.2 Speed

An additional important factor in the utility of a resource allocation algorithm is the "speed" of the allocation algorithm. Consider two algorithms that both produce acceptable, working resource allocations. If one algorithm can always compute an allocation at least as fast as the other allocation algorithm and both algorithms produce the same amount of resource slack, then the faster algorithm can intuitively be thought of as a "better" algorithm. However, faster algorithms might commonly need to tradeoff increased allocation speed for decreased resource slack. Assume there is a desired time t_0 for deploying a given set of application strings and t is the actual time used by the algorithm to produce an allocation for the set of application strings. Then the speed factor v for an algorithm can be expressed as a function of t_0 and t:

$$v = f_v(t_0,\ t)$$

The value of v will decrease with respect to t. Generally v can be normalized to be a real value between 0 and 1 ($0 \leq s \leq 1$) for mathematical simplicity.

For a simple implementation of the speed factor function, suppose n allocation algorithms are given that generate solutions with computation times $\{t_1,..,t_n\}$. Then, if $t^* = \max\{t_1,\ldots,t_n\}$ define v(i), the speed factor v for the i^{th} algorithm as:

$$v(i) = {}^{(t^* - t_i)}\!/_{t^*}.$$

Note that this implementation of the speed factor disregards the desired time t_0 for deploying a given set of application strings, but normalizes the speed factors so that the algorithm with the shortest time has the highest speed factor.

5.1.3 Defining Resource Utility

Based on the speed and efficiency factors discussed above, we established a resource utility function to evaluate the efficiency of a resource allocation algorithm. In distributed real-time systems such as ARMS MLRM, resource allocation occurs at multiple layers in the system hierarchy. It can generally be assumed that high-level resource allocators will not have detailed information about layers beneath them. Thus, as coarse resource allocations are performed at higher layers, refined allocations are then performed at lower layers in the system hierarchy. For example, in ARMS MLRM the *Infrastructure Allocator (IA)* only has resource information about the pools (not individual nodes in the pool) and allocates applications to pools based on its coarse grained information. A pool-level *Resource Allocator* then assigns applications (allocated to the pool by the Infrastructure Allocator) to specific nodes in the pool. The assumption is also made that a feasible allocation at a higher level implies feasibility at lower resource allocation layers for the sake of simplicity. The resource utility function discussed below is therefore applied to resource allocation algorithms at different layers of the hierarchy.

In order to trade off the computation speed and resource slack to gain the best possible performance, the resource utility function could be defined to be a weighted sum of the two factors:

$$UR = c_t v + c_{rs} s \qquad (9)$$

The variables c_t and c_{rs} are weights for the relative importance of computation speed and resource slack respectively. Generally, as experience is gained in using the system, the c_t and c_{rs} values can be adjusted to better reflect the relative importance of speed and resource slack when computing resource utility. It remains an open area for continued investigation to find experimentally optimal values for these weights.

The resource utility captured in Equation 9 is a starting point for evaluating the performance of allocation algorithms including the performance of the incremental mapping algorithm used in the Infrastructure Allocator and various bin-packing algorithms used at the pool level. If only a subset of strings can be deployed due to limited resource availability, then the resource utility alone may not fully indicate the quality of the allocation since the number of strings deployed might be different and thus produce a different system-level application utility as discussed in Section 5.1.1. In this case, one needs to look at both aspects of the utility and make a judgment on which aspect is more important. One possible simple approach to combine these two types of utilities into one is to define the overall utility U as a weighted sum of these two:

$$U = c_a UA + c_r UR \qquad (10)$$

where c_a and c_r are weights for the relative importance of these two forms of utilities to the system. We realize that developing a comprehensive way to capture the overall utility is a very hard problem and we'll try to address this in more depth in our future research.

5.1.4 Utility Metrics for Control

An important consideration for these systems when designing online metrics to be used with control functions is the lack of accurate and timely information. Whereas offline system evaluation is generally performed with full information about system behavior after the fact, online performance and decision procedures rarely have the benefit of full system oversight. For instance, the controller would need to know which resources it can (currently) deploy strings on, but the transmission of this information to the controller may be delayed. Similarly, the controller may not be instantaneously aware of job and/or string failures. Any design of a control architecture would need to take these factors into account.

5.1.5 The Gate Test 4 Metrics

As an aid to the reader, we present here the GT4 metrics from the GT4 CONOPS. We designed our utility function due to perceived deficiencies in using these Metrics to guide control decisions. These concerns are discussed in Subsection 5.1.5.3. Note that the Warfighter Metrics are both evaluative metrics and are not intended for use as a run-time measure.

5.1.5.1 Metric 1

An important concept in the definition of the GT4 Metric 1 that is not expressly addressed in our utility function designs is the concept of "most important strings." GT4 Metric 1 was designed to measure "the proportion of time that all of the highest priority applications strings remain operational." We understood the most important strings to be the strings which are assigned the highest importance values. The equation given for Metric 1 in the GT4 CONOPS is:

$$M1 = \frac{\sum T_{up}(S_{MAX})}{T * S_{MAX}}$$

This function is computed over a run of system operation. It is the sum of the up time of the most important strings divided by the number of most important strings and the run time of the experiment.

5.1.5.2 Metric 2

Similar to Metric 1, Metric 2 is the "average total warfighting value of operational application strings". Metric 2 can be thought of as analogous to a coarse-grained version of our application utility divided by the run-time of the experiment.

$$M1 = \sum \left(\frac{T_{up}(S)}{T} * WFV \right)$$

Over a run of system operation, Metric 2 is sum of the proportions of time the strings are running weighted by their warfighter values.

5.1.5.3 Perceived Deficiencies in the Gate Test 4 Metrics

The two performance metrics are intuitively designed to measure the ability of the system to allocate 1) sufficient resources for deploying critical strings and 2) resources to maximize the sum of the warfighter values of the deployed strings. Unfortunately, the two performance metrics do not explicitly account for the ability of the system's strings to process other less critically important jobs when evaluating system performance. For instance, during situations of resource contention, if a less-critical string processes jobs at a sufficiently low frequency or aperiodically such that if between a string's failure and recovery, no jobs are dropped, then there should be little or no penalty reflected in the performance evaluation of the system due to the short-term downtime of this string. This is because the uptime of low frequency or aperiodic strings is not necessarily proportional to the number of job successfully processed by the string. On the other hand, the GT4 evaluation metrics as defined would most closely match the intended purpose when used with strings with very high job processing rates. The reported uptimes of these strings would be highly correlated with the proportion of jobs processed by these strings.

Because there is a penalty reflected in the performance evaluation of the system due to the short-term failure of a less-critical string when no jobs are dropped, there is no benefit to prioritizing less-critical string recovery based on job scheduling.

Naturally, the most critical strings should always be allocated resources no matter there job scheduling. However, if the system is ever in a situation where there is so much resource contention that resources cannot always be allocated to even the most critical strings, then the scheduling approach outlined above for less-critical strings could also be applied to the most critical strings in order to generate some level of warfighter value from the system.

The resource allocation system should be designed to give higher string recovery priority to strings with impending jobs to achieve actual warfighter benefit. Therefore, we propose that evaluative measures for system performance should account not just for the ability of strings to be allocated sufficient resources, but to explicitly account for string failures during performance evaluation only when jobs are not processed. In the first part of this section, we discussed our proposed real-time utility measures not only as an off-line method for evaluating system performance, but as part of an online system performance measure that the resource allocation system can use to guide its allocation decisions toward improved warfighter value.

With the currently defined performance metrics, it is natural to define the online control metrics as an approximate numerical derivative of the offline evaluation function. With this relationship, if the online control metric reports a high level of value at an instance in time, then the offline evaluation metric should accumulate a high level of value due to the operations of the system at that time. This can be done in several ways. Using the currently defined evaluation metrics, the rate of increase of the offline metrics is equal to a (possibly weighted) sum of the current deployed strings. If one were to use a performance evaluation metric based on the accumulation of rewards due to successful job completions, the online control metric should be a (possibly weighted) function of which strings can successfully process their next jobs in order to better capture the online ability of the strings to accomplish their tasks.

Utility Measures of System Performance:

The performance metrics used by the system should naturally be accumulative measures of system performance. That is, rather than instantaneously attaining high levels of system performance intermittently, it is generally preferable to attain a lesser level of instantaneous performance as long as the average warfighter value is higher. This property is well reflected in the currently defined GT4 performance metrics, which are essentially sums of the integrated performance of individual strings. However, if one were to take a job-based view of system performance evaluation, one would need to measure system performance as the weighted summation of successfully processed jobs rather than the integral of system uptime.

As discussed above, the method used for the evaluation of system performance has a significant impact on how resource allocation decisions are made when controlling the system. Whereas the evaluative measure for system performance is computed offline, dynamic resource allocation systems, such as the MLRM, generally need an online performance measure that can be used to evaluate the health of the system and decide on the best course of action during run-time. For instance, when deciding whether to take a resource allocation action, a system controller should be able to estimate the possible benefits and penalties associated with those actions. A resource controller should be able to decide online when the system is performing insufficiently and that it needs to take action to improve performance. Similarly, when deciding how to allocate resources to strings, a resource controller would need to know the relative benefit of deploying the strings on the various resources. Because the system performance measures are inherently defined as integral (summation) equations, it is natural to define the control performance function as a kind of numerical derivative (or instantaneous rate of change) of the offline evaluation functions so that the controllers can have information about instantaneous system performance.

Moving Forward

With the above points we have raised regarding performance metrics for evaluation and control, we believe that:

Evaluative measures of system performance should account for the ability of strings to accomplish its jobs, not just that it has sufficient resources to run.

Whereas accumulative and/or average measures of performance are used to evaluative system performance, control decisions should be made using instantaneous measures of performance that measures the rate of change of accumulated utility.

Evaluative measures of system performance should use information that is accurately available to the control system in a timely manner so that the control system can have sufficient information about system health to make the proper control decisions.

5.2 Hierarchical Control for Dynamic Resource Management

This section discusses our design for the hierarchical feedback control DRM system approach for the ARMS Multi-Layered Resource Management (MLRM). Several operational aspects of the control system introduced in this section are left intentionally vague at this time to more succinctly convey a high-level overview of our design of the control system. We designed this DRM system so that various local resource management algorithms can be used at the various levels of the hierarchy to tune local behavior. Details about our algorithm designs are discussed in Subsection 5.3. In this following subsection, these algorithms are compared through the use of a Matlab/Simulink simulation package that we developed and several of the more difficult implementation issues are identified as areas of continuing investigation.

5.2.1 Control System Overview

Our goal was to design an ARMS MLRM in which computational resources are allocated so that deployed application strings perform their desired operations with maximum utility. In general it may occur that some resources, once allocated, may become inaccessible due to resource contention and unpreventable system behavior such as hardware failures. When these situations arise, the resource control system should be able to automatically and dynamically manage resources in order to ideally maximize the system's application utility. This subsection describes a multi-level hierarchical control system to dynamically manage resources in the ARMS MLRM based on a system-mission-string decomposition and motivates the architecture for this control system in the framework of application utility maximization.

At initialization, strings are deployed using services provided by a resource allocation algorithm such as incremental mapping or multi-dimensional bin-packing. After initialization, the control system then dynamically and continuously manages resources in the system under changing conditions including failure conditions such as node and pool failure. The hierarchical resource control system interacts with the system at three levels: system, mission, and string.

At the highest level, the system controller has the ability to trigger system-wide full reallocation or reconfiguration of resources involving coordination between multiple missions. There is expected to be a relatively high cost associated with performing such a system wide action, so global allocations should only be performed when the cost of these actions can be justified by a sufficiently large gain in application utility or when no other effective actions are available. These system level reallocations or reconfigurations may by driven by command decisions, or triggered as an automatic response to changes in the system that could not be mitigated by lower level controls.

At the mission level of operation, the mission controller manages the behavior of strings to maximize the local mission-level application utility. Possible actions that could be taken at the mission level include moving strings/substrings operations between pools after failures or in response to increased load, starting or stopping strings, or killing and generating replicas to attain proper levels of fault tolerance. Due to its inherent difficulty, we temporarily disregarded the problem of killing or generating replicas. Because control actions taken at the mission level can have wide-ranging impact, multiple mission level controllers will typically have to coordinate their actions with the system level controller or even amongst themselves.

The application utility functions defined in Subsection 5.1.1 are computed at the lowest level via the summation of string level utilities. We therefore started from the proposition that string level control actions are natural to be used at the lowest levels of the control hierarchy. (If this granularity proved inadequate, we would have provided a finer granularity for control actuation (e.g. by controlling individual applications).) Strings can locally and independently tune their operations in order to locally maximize their application utilities by adjusting their controllable parameters such as throughput and quality factors. There is little or no cost associated with this low-level fine tuning, so the strings are allowed to perform these actions freely bounded only by their resources limits and desired operational ranges.

Since controllers at all three levels are simultaneously performing their actions at runtime, coordination between them is thus critical and essential in order to take appropriate action at the right time and at the right level while avoiding conflicts between controllers at different levels. We proposed a fundamentally bottom up approach for the ARMS MLRM design that we believed to be suitable for managing the type of operations and applications in the system. Additionally, we augmented this bottom up approach (to the dynamic management) with a top-down information passing policy so that the system user and/or high-level controllers can prompt the lower level controllers to take local actions.

With our bottom-up approach for the control system, when a low-level string controller is unable to markedly improve or maintain its desired application utility, it would communicate this to its mission controller. This triggering event drives the mission controller to perform mission-wide adjustments of its strings in order to attempt to improve the string utility if the mission controller deems these actions feasible. When the mission controller is unable to improve its application utility (when it is actively seeking improvement`), it would similarly communicate this to the system controller. This signal may prompt the system controller to perform system-wide reallocations in order to attempt to improve the mission utility. The advantages of this bottom up approach include rapid local responses to changes in system behavior, minimized communication cost between control levels, and scalability of the solution.

Throughout all operations in the ARMS MLRM, all operations have an application utility that expresses the relative ability of the system to perform its desired behaviors. Along with the Warfighter Metrics, this utility evaluation should be maximized over the course of system behavior and in the face of adverse conditions such as equipment failures.

Now that an overview of the multi-level control system has been presented, a more detailed design of the three levels of the control system is presented in the following subsections.

5.2.1.1 System Level Management

As described above, the highest level of the resource control system performs system level reallocations and reconfigurations when the mission level controllers are unable to sufficiently improve system performance and it is estimated that the utility of the resulting allocation minus the associated cost is greater than the utility of the current allocation. To illustrate this in a more formal manner, suppose x is the current resource allocation state, and x' is the new allocation state if a candidate reallocation R occurs. This reallocation transition is denoted as $x \rightarrow_R x'$. Let $EU(x)$ denote the estimated utility resulting from maintaining the current state x, let $EU(x')$ be the estimated utility for the new state x', and let $EC(R)$ be the estimated cost of performing the reallocation R. Then, if $EU(x') - EC(R) > EU(x)$, there would be an overall gain in utility if the candidate reallocation R were performed. The $EU(x') - EC(R) > EU(x)$ test is known as a threshold passing test and it is used to decide if the net utility gain $EU(x')-EC(R)$ passes the $EU(x)$ threshold to signal that there would be a net gain in utility by performing the reallocation R.

We originally conceived that system level resource management operations would typically occur after dramatic changes in the system such as multiple node and pool failures, or when the system is asked to reconfigure itself under changing mission priorities.

As a simple example of how we originally conceived the system level controller could work, suppose nine critical strings are running in a system with three pools. If one of the pools were to fail and there are three strings partially deployed on that pool, then those three strings would no longer be running and there might be an estimated utility of $EU(x_6)$ of solely maintaining the remaining six strings on the two operational pools. Suppose one of several possible reallocations could be performed that the three failed strings (or more precisely their substrings on the failed pool) are redeployed to the other two pools such that this control action would have a resulting estimated utility of $EU(x_9)$. It is possible that in order to perform this reallocation, it is necessary to temporarily halt the operation of some of the six running strings on the operational pools with an estimated cost of $EC(R)$. If $EU(x_9) - EC(R) > EU(x_6)$, then it would be advantageous to perform the redeployment, but if the six operational strings are so critical that the reallocation cost would become so great that $EU(x_9) - EC(R) < EU(x_6)$, then it would be better to maintain system operation with only the six strings operating. Although there might be other possible control actions in order to redeploy the failed substrings, this example is given as a demonstration of how a change in utility from a deployment might be outweighed by the cost of performing a control action.

Any number of possible algorithms could be used to compute candidate reallocations could be used, such as an implementation of multi-dimensional bin packing which was investigated by Lockheed-Martin, or a modification of the incremental mapping method being developed by JHU APL. However, it is necessary that when choosing a candidate reallocation, there be some method of estimating the reallocation utility, *EU(x')* and the reallocation cost, *EC(R)*. We conceived that a possible methods for finding candidate reallocations would operate in an online manner such that during the course of system operation, the search algorithm could continually search for and maintain a list of the "best" found candidate in memory. The internal best candidate found would be continually available to the high-level control system such that if the high level control system ever decides that EU(x')-EC(R) > EU(x), then the reallocation is performed. In this manner, the high-level control does not necessarily need to wait for the allocation search algorithm to cease operation or find the optimal solution, but the high-level controller only needs to wait for a reallocation that is "good enough." We ended up not implementing this specific approach, but we did use a related approach based on the estimation of possible resource allocations.

We realized that the precomputation method for finding candidate reallocations might not always be viable under sudden and significant changes of state such as those occurring due to battle damage. During these situations, the controller should have a precomputed static default deployment to use as a failsafe, baseline mode.

5.2.1.2 Mission Level Adaptation

For mission level control, the underlying philosophy is that instead of performing wholesale reconfiguration of the system, the mission level controllers are intended to actuate discrete control actions to adjust the performance of the mission and the performance of all the strings within the mission. As an example of this philosophy, to revisit the three pool example from Section 5.2.1.1 where a single pool fails, instead of reallocating all of the three affected strings to the other pools simultaneously, it would be feasible for the mission level controller to push one of the failed strings onto one of the other operational pools. Note that in this example, it is assumed that all of the strings belong to the same mission.

Control actions which we conceive as being feasible at this level include the starting or stopping of individual strings, killing or generation of replicas, the moving of individual strings or substrings between pools or nodes, all within a single mission. It is generally assumed that the control actions at this level have a lower cost and impact than the combined operations that occur at the system level controller. Similarly, it is assumed that this mission level of the control hierarchy operates on a faster time scale than the system level controller.

We conceived that at the mission level of operation, the controller would have a finite set of known possible control actions $\{u_1,\ldots,u_n\}$. Similar to the system level controller, it is assumed that each of the control actions could have an associated (possibly 0) cost $EC(u_i)$ (for $i \in \{1,\ldots,n\}$).

Suppose at time t the system is at state x. For every control action u_i taken at x there is a corresponding resulting state x_i such that $x \rightarrow_{ui} x_i$. As for the system level management, if $EU(x_i) - EC(u_i) > EU(x)$, then the utility of the reallocation would be more beneficial than the utility of the original allocation. Therefore, in this case it would be advantageous to take the control action u_i in order to maximize the overall estimated utility. In the case that there are multiple control actions that result in $EU(x_i) - EC(u_i) > EU(x)$, the correct control action would be to choose the u_i that maximizes $EU(x_i) - EC(u_i)$.

Although we did not design the system in this manner, if there are multiple missions in the system, mission level controllers may need to coordinate their actions, potentially through direct coordination with the higher level system controller. This need for coordination was perceived as possibly being of interest because actions taken at the mission level can have wide-ranging impact. However, during our simulation experimentation, we found that we were able to significantly improve our system over the baseline without this extra complication.

5.2.1.3 String Level Tuning

At the lowest level of the resource allocation control systems, string operations are dynamically and continually tuned using a gradient descent algorithm to locally maximize their application utility. Tunable parameters that locally influence a string's application utility include quality and throughput. Mission requirements generally determine the ranges of these tunable parameters and they need to be captured in the string specification, e.g. AIM or its evolving equivalent. Other string-level QoS attributes (timeliness, availability, etc...) cannot be directly controlled at this (string) level. A string's application utility is directly proportional to both its quality and throughput. The control actions taken at the string level to adjust a string's quality and throughput generally have little or no cost associated with them beyond the cost/utility observed directly through the application utility function. Therefore, the fine-tuning of these parameters can be allowed to occur regularly as long as the resource consumption is within its limits and the string stays within the desired operational range. Note that timeliness is indirectly influenced by manipulating quality and throughput, so the over-actuation of quality and throughput may adversely affect application utility by decreasing a string's timeliness.

Generally, if a string is unable to maintain a sufficient amount of local utility, it would be able to transmit this difficulty to its mission level controller. In this manner, the control can be properly transferred to a higher level controller in order to maintain the performance of the system. Similar control transfers can also occur between the mission level and the system level in this bottom up control approach.

5.2.2 Implementation Plan for the DRM Control System

Now that an overview of the MLRM control system has been presented in Section 5.2.1, this section fills in more details in the above high-level discussion by showing our initial, albeit still simplistic, implementation plan for the MLRM control system. We used this design to explore key details of how the control system could operate in practice, and as a means to identify several key aspects of the MRLM control system which, as properly addressed in future work, are needed to enable and improve the abilities of the controllers. First, in Subsection 5.2.2.1, we discuss how we planned on implementing the utility estimation functions from Section 5.1. Then, in Subsection 5.2.2.2, we discuss how we initially conceived the controllers should operate and coordinate their actions. Subsection 5.2.2.3 presents a control example for the operation of this planned system.

5.2.2.1 The Implementation of Utility Estimation Functions

As discussed in Section 5.2, the operation of the hierarchical control system is predicated on being able to estimate the utility of resource allocations that may result due to its possible control actions. Several omitted items in our control system description in Section 5.2.1 are the details of how we planned for the various level of the control system should 1) obtain information about system behavior, 2) determine what its feasible control actions are and 3) determine how to actually estimate the utility of those actions.

Observations Of System Behavior

The various controllers in the MLRM system determine their appropriate control actions by processing information about the current "state" of the MLRM system. The controllers can obtain their information through direct observation of behavior in the system via various monitors in string applications or on nodes, by communicating with other elements of the MLRM infrastructure such as the pool managers and bandwidth broker, or by querying the RSS at different levels.

Although there may be more optimal sets of information that the controllers should obtain other than what we initially expected, our expectation is that this knowledge will become more intuitive and apparent as experience is gained in working with the system. However, we were confident that the information specified here for the controllers to be able to access is a reasonable first step towards the ultimate goal of designing an effective control system for the MLRM, which is vastly superior to, more flexible, and better organized than current (static) practice.

To start, the low-level string controllers need to access information about the string's local throughput, quality, and timeliness through various string and application level monitors. In addition, a string failure monitor should push such failure events to the string controller (likely through an RSS callback). These observations of string behavior are fundamental to the computation of string application utility in order to maintain the desired operation of the string. Therefore, this information should be accessible to the string controller. From these observations, the string level controller can compute the string's application utility and push this value to the RSS so that the mission controller can obtain this information to compute its mission level application utility.

Because the mission level controllers coordinate the behaviors of sets of strings, the mission level controllers should be able to obtain the Application Utilities of the strings that belong to the mission. This information can be obtained either through push or pull from the RSS (as mentioned above the string controller pushes this information into RSS). Therefore, the mission-level controllers are able to compute their mission's application utility and push that to RSS for a system level controller to consume. Additionally, the string-level controller needs to be able to transmit information directly to its mission-level controller upon the detection of software failures in the string.

Because the mission level controllers take control actions with potentially system-wide implications, they need to obtain resource information, including information about pool and communication failures, from the Bandwidth-Broker and pool managers via RSS. Additionally, the mission-level controllers need to obtain information about their ability to allocate resources for control actions potentially performed by the mission-level controller. This information can also be obtained by querying pool managers and bandwidth broker.

At the highest control level, the system level controller needs to access mission level application utilities. Similar to the mission level controllers, information about pools and communication links including various failure events also need to be made available to the system level controller. A schematic of how information is passed between the system layers can be seen in Figure 29.

One of the key challenges in designing the hierarchical control system is the coordination and communication between different control levels and the organization of possibly multiple triggering events in the same controller. As mentioned in Section5.2, the coordination between controllers at different levels will occur in a bottom-up manner within a top down policy frameworks that changes in the system will be handled at the lowest control level if possible and the higher level controllers supervise the local behaviors of the low-level controllers.. In this manner, a user of the MLRM control system has the ability to give high-level resource management directions while avoiding the need to dictate low-level operations. Within a controller, multiple triggering events can be received and the controller needs to figure out the right action based on the type and timing of these events. For example, during a pool failure scenario, three events could be reported to the mission level controller: a dropping of mission Application Utility, string failure, and pool failure. These events could arrive at different times and we expect failure detection would be generally faster than the detection of utility drop. As a general approach, we propose that failure events take precedence over making of observations, and larger scope events take precedence over smaller scope events. Similarly, failure-recovery operations and command directives pre-empt all other control system operations, and command directives pre-empt failure-recovery. Therefore, in this case the pool failure event will be handled by the mission controller instead of the other two, assuming they are received within a "grace" period. This "grace" period could be the worst-case failure detection at the mission level. For example, if the "grace" period is set at 2X the effective heartbeat for a computing pool, then the controller will wait at least that long (to allow other events to come in) after receiving the first event. In addition, a control action could be preempted if the controller believes the later action can solve the cause of the problem and the earlier action won't. For example, a string failed in the system and for some reason the utility drop was first detected by the string level controller. Within the grace period, the string failure was not detected and action was taken to tune the string locally. After that control action, the failure was received by the controller. It determines that failure was the root cause of the utility drop. At this point the tuning is stopped and string failure is handled.

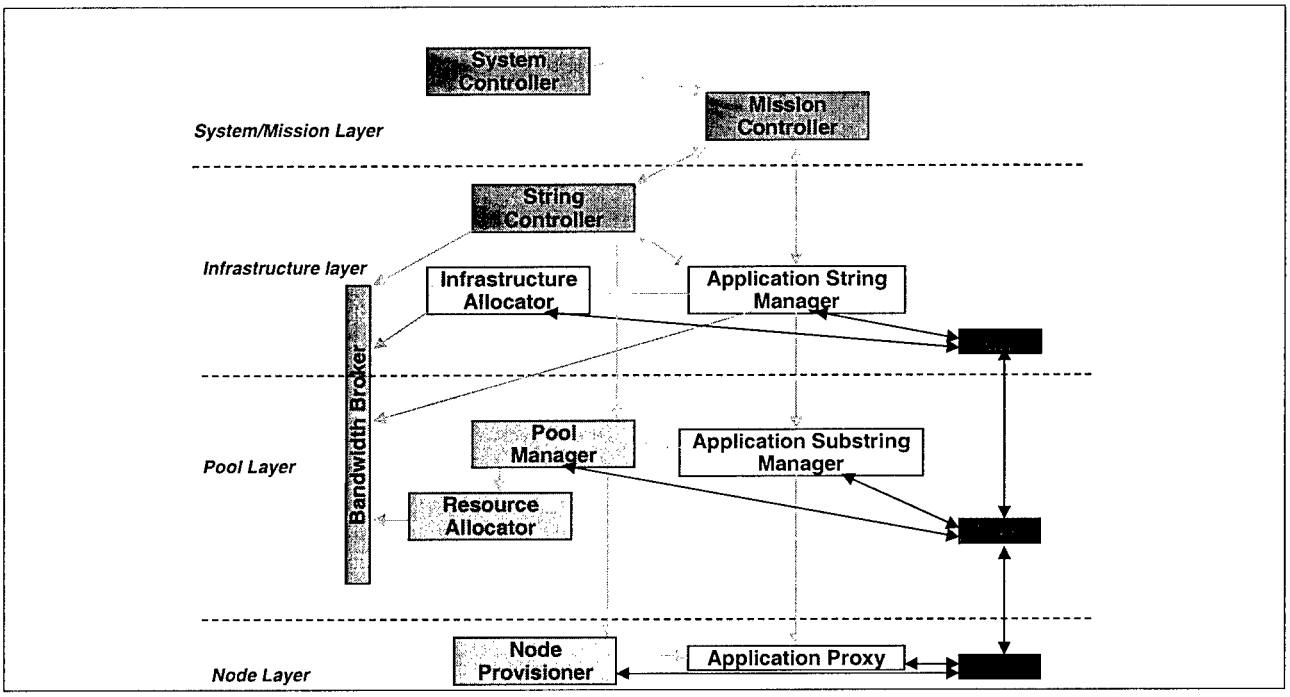

Figure 29 Control communication paths between control system layers

Determining Feasible Control Actions

Now that we have described to a first order of detail how we planned for the multi-level controllers to obtain and share information about the ARMS MLRM, we describe how the controllers at the various levels determine what their feasible control actions are.

As described in Section 5.2.1.3, the low-level string controllers have the ability to tune their local quality and throughput in attempts to maximize their local control actions. Because the actions of the string controller are localized, it is generally always feasible for the string controllers to manipulate these controls as long as it stays within its desired operational ranges. As mentioned in Section 5.2.1.3 these ranges are captured in the string specifications (and potentially updated dynamically as has been demonstrated in the PCES program) typically with maximum and minimum limits. The mission level controllers are generally allowed to generate and/or kill replicas and move substrings of their strings. For the sake of simplicity at this stage of development, we purposefully deferred the ability of missions to generate and/or kill replicas. Therefore, the sole method currently considered for mission-level controllers to perform control actuations is to move substrings of their strings between pools. The mission level action is feasible if the action does not impact other missions running in the system. For example, in order to test if moving a pool substring is feasible, a mission level controller would first have to query the pool manager of the destination pool for the substring to determine if the destination pool has the resources to accept the substring. If the destination pool can accept the substring, the mission-level controller then needs to query the bandwidth broker to verify that the bandwidth broker will be able to allocate a sufficient amount of resources to move jobs along the string potentially both to and from the destination pool. If the destination pool can accept the substring, and the bandwidth broker can allocate sufficient communication resources, then it is feasible for the mission level controller to move the substring. At this point the whole string will be stopped, the substring will be killed and redeployed in the destination pool, and then the string will be restarted.

At the highest level of operation, the system level control performs reallocations or reconfigurations of the system missions and strings. Similar to control actions performed at the mission level, the reallocation control actions are generally always feasible as long as the bandwidth broker and various pool managers verify that there are sufficient resources in the system to perform the reallocation. If there are no feasible actions for the system level controller to perform, the system level controller could exercise the option of reverting to a pre-computed failsafe resource allocation mode, which although not optimal, may permit the system to continue operation and provide a level of utility to the user.

Estimating the Utility of Control Actions

Now that we have described how we planned for the various system controllers to obtain their information and determine the feasibility of their possible control actions, we describe how the controllers decide what their control action should be. Central to determining the correct control action is the problem of estimating the change in utility due to a control action. That is, for any possible control action R such that $x \rightarrow_R x'$, values need to be assigned for EU(x'), EC(R) and EU(x). To start, we deferred the problem of estimating control cost (EC(R)) to aid in the simplicity of our initial design. We found in our later simulation implementations, that it was not necessary to account for control cost to achieve higher levels of performance in the system.

For our planning at this time, for all possible control actions in the MLRM system implementation scenario we describe in this section, it needs to be determined if EU(x') > EU(x) in order to take the control action R such that that $x \rightarrow_R x'$. The problem of obtaining a meaningful and easily computable utility estimation function (EU(•)) is, in our opinion, far and away the most difficult problem associated with the design of the MLRM control system. Invariably, like the EC(R) calculation, a high level of contextual system knowledge is required to develop a meaningful utility estimation function. Ideally, the relative ordering of the estimated utility of two allocation states should, as often as possible, be identical to the relative ordering of the observed application utility of those two allocation states. That is, for as many (x,x') pairs as possible, we would want an utility estimation function EU(•) such that EU(x') > EU(x) if and only if UA(x') > UA(x). Therefore, in order to implement the proposed hierarchical control system, we propose a class of simple utility estimation functions for the string, mission and system level controllers based on estimates of two strings' expected timeliness, quality and throughput. Unfortunately, there are inherent computational difficulties associated with distributed control problems that make it difficult to exactly predict the effect of multiple, possibly uncoordinated distributed control actions on, for example, local string timeliness observations. This difficulty would also make it difficult to accurately predict or estimate the relative ordering of the application utility of two different allocation states. The proposed alternative (below) to the timeliness estimation is a meaningful and useful first step towards the ultimate development of a "better" or even "best" utility estimation function.

Our expected timeliness estimation computation is based on a method for estimating the expected delay of jobs in a string. Naturally, jobs in a string should have ideally as small an end-to-end delay as possible. This sentiment is expressed in the definition of the timeliness factor in Section 5.1.1.1 above where the timeliness factor decreases as the end-to-end delay of jobs in a string increases. (For the sake of simplicity, a job's end-to-end delay in a string is simply called delay from now on unless explicitly noted otherwise.) Because the application utility of a string is proportional to timeliness, then the application utility also decreases as delay increases. As an example of this scenario, consider a string with hard real-time jobs where each job has a timeliness factor of 3 if the job is completed in less than 4ms and 0 otherwise. Although the timeliness factor in this example does not depend linearly on delay, timeliness does decrease as delay increases.

Besides depending on timeliness, as discussed in Section 5.1.1.5, in the application utility function, utility is directly proportional to the quality factor q and the throughput Th. Therefore, as a simplistic approximation of a string's estimated utility, EU_s we propose to use Equation 11 where the estimated utility is inversely proportional to d, the delay factor of jobs in a string and directly proportional to quality and throughput:

$$EU_s = qTh / d \qquad (11)$$

As described above, the utility estimation function in Equation 11 encodes the intuition that strings with high throughput and quality should have a high utility, while strings with a high delay should have a low utility. Note that in Equation 11 there is an implicit simplifying assumption that the delays of jobs in a string are independent of quality or throughput. This is generally not the case, but these effects are ignored for now for the sake of simplicity. There is also the implicit simplifying assumptions that average delay can be used as a proxy for specific delays encountered; this is not generally the case either, but hopefully is indicative enough of the trend to make the simplifications and tradeoffs meaningful.

In order to develop an estimate of observed delay for jobs in a string, suppose a string is deployed with substrings sequentially in Pools 1 through P such that upon querying the pool manager for Pool i, the mission controller is told by the pool manager that jobs in the string belonging to the string controller will have an estimated expected computation time of dp_i when passing through the substring in Pool i. The ability to compute the estimated expected computation time of jobs on a substring deployed on a pool is not a current ability of a pool manager, but this ability would need to be added to implement the control system discussed in this paper. Similarly, suppose that a string passes through inter-pool communication links 1 through L such that upon querying the bandwidth broker, the mission controller is told that data being processed for the jobs in the string have an estimated expected transmission delay of dl_i when passing through communication link i in the string. The ability to compute the estimated expected data transmission delay for jobs transmitted between pools on a string is not a current ability of the bandwidth broker, but this ability would need to be added to implement the control system discussed in this paper. Due to the sequential data processing in the string in the P pools and L inter-pool links the string is deployed in, the jobs in the string would have an overall end-to-end delay to traverse the string proportional to:

$$d = \sum_{i=1}^{P} dp_i + \sum_{i=1}^{L} dl_i \qquad (12)$$

The delay factor of Equation 12 can then be used as in Equation 11 to estimate the utility of a resource allocation to a string. As the delay measures returned by the pool managers and bandwidth broker in Equation 12 is at best an estimate of system behavior, we do not argue that Equation 12 is the "best" or most accurate method for approximating the end-to-end delay experienced by jobs in the string, but we propose that this is a reasonable, easily computable first-order approximation. Note that Equation 12 is only concerned with an estimate of the expected end-to-end delay of jobs in the string rather than the absolute delay of individual jobs.

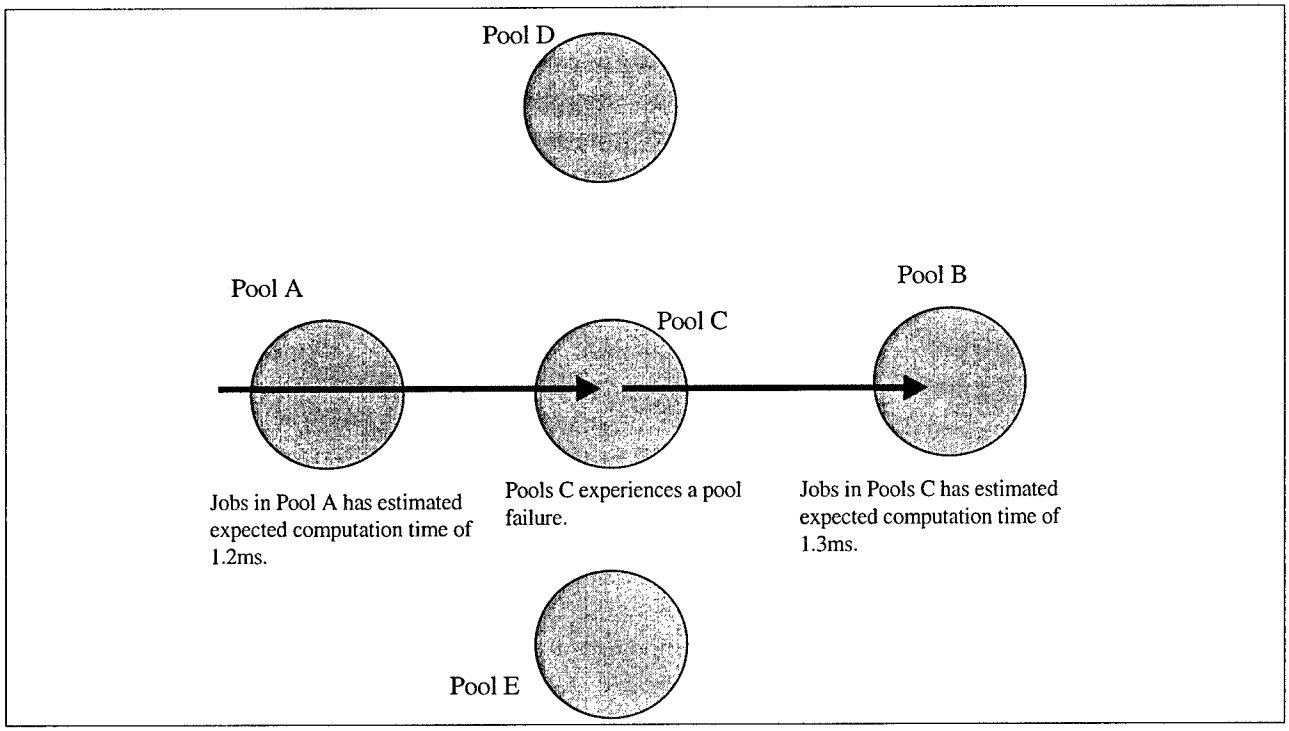

Figure 30 Initial deployment of a string.

To compute the mission level estimated utility, suppose a mission is composed of a set of strings $\{s_1,\ldots,s_m\}$ with relative importance values $\{w_1,\ldots,w_m\}$. Suppose that any string s_j has an estimated utility of EU_{sj}. Therefore, using the summation approach to compute mission level utility from string-level utility used in the definition of application utility, EU_m, the estimated utility of the mission is

$$EU_m = \sum_{j=1}^{m} w_j EU_{sj} .$$

Similarly, if the system comprises a set of missions $\{m_1,\ldots,m_n\}$ with relative importance values $\{w_1,\ldots,w_n\}$. where EU_{mk} is the estimated utility of mission m_k, then let EU_{sys}, the estimated utility of the system configuration be:

$$EU_{sys} = \sum_{k=1}^{n} w_k EU_{mk} .$$

As an example of how the estimated utility function can be used to compute the relative desirability of mission level control actions, consider the system in Figure 30 that comprises one string deployed sequentially across Pools A, C and B. According to the substring managers of the strings, jobs in the substring deployed on Pool A have an estimated expected computation delay of 1.2ms and jobs in the substring deployed on Pool B have an estimated expected computation delay of 1.3ms.

Suppose a pool failure occurs in Pool C and the mission controller can redeploy the pool substring on Pool C to either Pool D or Pool E as seen in Figure 31. Suppose that, according to the respective pool managers, jobs in the substring in Pool C would have an estimated expected computation delay of 1ms on both Pool D and Pool E.

Suppose also that it is communicated to the mission controller from the bandwidth broker and pool managers that the transmission of the data for jobs from Pool A to Pool D would take an estimated 0.6ms, the transmission of data for jobs from Pool A to Pool E would take an estimated 0.5ms, the transmission of data for jobs from Pool D to Pool B would take an estimated 0.4ms and the transmission of data for jobs from Pool E to Pool B would take an estimated 0.7ms. For the sake of discussion, assume for both redeployments that the string controller maintains qTh = 1.

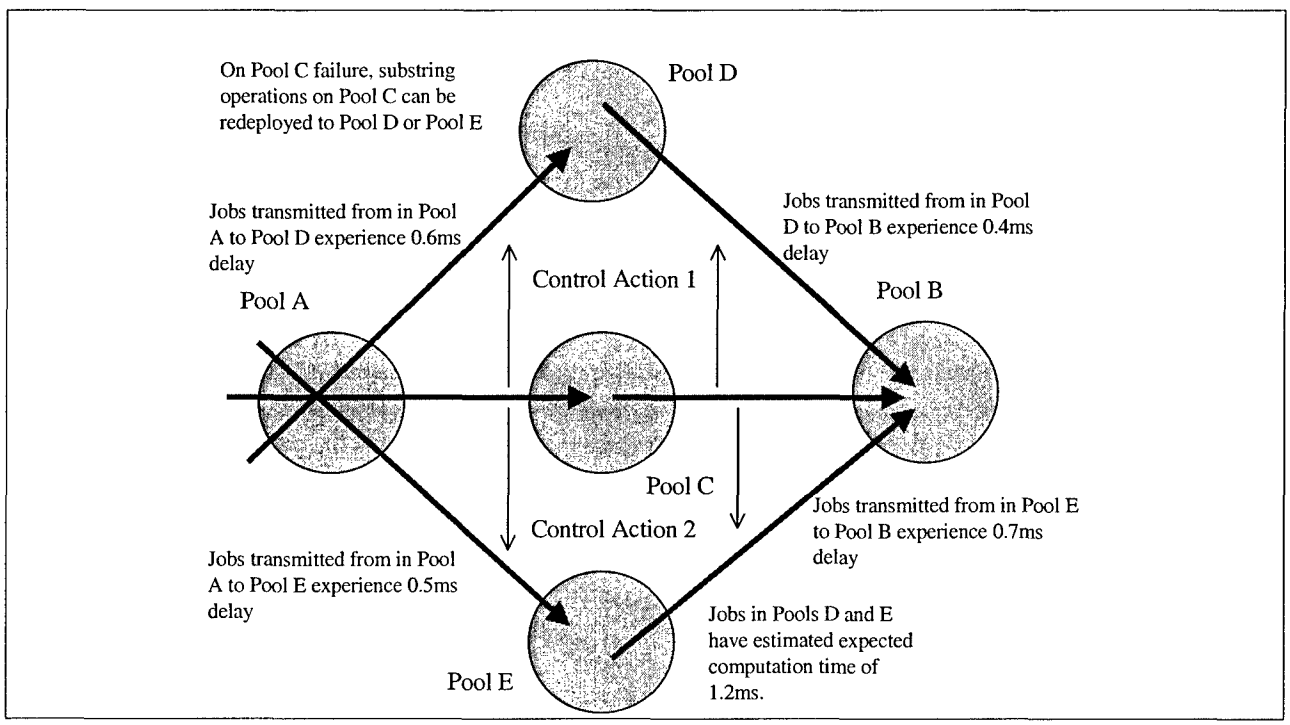

Figure 31: Two possible redeployment options for a string.

To compute the estimated utilities of the two strings in their current deployment state, the expected end-to-end delay experienced by jobs in the string if the Pool C substring is moved to Pool D is 4.5ms and the expected end-to-end delay experienced by jobs in the string if the Pool C substring is moved to Pool D is 4.7ms. Therefore, EU(String w/ Control 1) = 1/4.5 = 0.22 and EU(String w/ Control 2) = 1/4.7 = 0.21 by Equation 12.

With this simple analysis of the two possible control actions, the mission controller would attain the highest utility if the Pool C substring were to move to Pool D.

Recall that the estimated utility measures are not used to estimate the absolute values of the system's application utility due to various control actions, but only as a easily computable method for approximating the relative ordering of the application utilities of various deployments. Small difference in the estimated utilities of the two control actions give an indication that it may be difficult to predict the relative ordering of the two deployments accurately. However, it is difficult to know how far apart two estimated utility values need to be in order to have a high level of confidence that one control action is clearly better than another, as was seen in the example in Figure 30 and Figure 31. Furthermore, there might be different control costs associated with the two proposed control actions, which although disregarded in this version of the control system, might be sufficient enough to skew the analysis of the "correct" control action. These two issues will be addressed in future versions of the control system through refinements as experience is gained with implementation of system and the control architecture.

5.2.2.2 Operational Overview

Now that we had organized and had a preliminary design for several of the most important details associated with the implementation of the MRLM hierarchical control system, we considered how the above implementations of the utility estimation function, the feasibility testing operations and information sharing between the system components can be combined into an implementation of the MLRM control system. The goal of this section is to demonstrate how the initially planned control system would operate, with special attention paid to the interactions between the MLRM system and the multiple control levels. We begin by discussing how the control system operates at initialization and in the absence of any system degradation. Then, we discuss how the control system operates when behavior in the system begins to decay. As a motivating example, we discuss how the control system would operate in the occurrence of a pool failure.

System Initialization and Normal Operation

As described above, at initialization, missions and strings are deployed in the system using the IA with the incremental mapping resource allocation algorithm. Unfortunately, the current IA implementation "over packs" strings even if there are insufficient system resources to accept a given deployment. This deficiency in the current allocation algorithm implementation needs to be addressed as part of the architecture enhancement so as to have a feasible initialization resource allocation strategy with acceptable levels of utility over more general configurations. This would be useful not only for initialization purposes, but would aid in the ongoing operation of the control system.

After system initialization and under normal operating conditions, the controllers perform no control actions and operate in "standby mode" such that at periodic time intervals, not necessarily synchronized, all controllers in the MLRM system sample their observed application utilities. The controllers retain the mean of their sampled application utilities in memory. The local application utility is a measure of how well the system has been performing in the immediate past. Therefore, if a controller has statistical information about its past application utility measurements, the controller can then detect a degradation in performance if the controller's observed application utility deviates sufficiently from its observed mean application utility. Upon the detection of a sufficient degradation in its observed application utility, a controller transitions from "standby mode" to "active mode" and takes actions to raise its observed application utility as discussed below.

As mentioned in Section 5.2.1.2, in addition to retaining the mean of their mission level application utilities, the mission level controllers could access the mean of their strings' individual application utilities. Similarly, at the system level, the system controllers retain the mean of the system level application utility and have access to the means of the application utility of missions in the system.

An advantage of using a controller's mean application utility measurement is that it avoids the difficulty of retaining all past application utility measurements in memory in order to detect a change in application utility. Naturally, the controllers observe fluctuations in their application utility, so a major problem is for the controllers to detect when a deviation in the observed local application utility from the observed mean warrants them to take action. A simple solution to this problem is implemented in the next subsection when controller operations are discussed.

When designing the initial DRM plan, it is an open problem to determine how often the controllers should sample the application utility. As a general rule of thumb, application utility samples should be taken as often as the controller might need to update its control action. We intended that optimum values for these parameters would be determined as experience is gained with exposure to system operation, but we started with the supposition that all controllers sample their application utility once a second or a sampling rate of 1Hz.

Determining When and How to Take Control Actions

String Level Controllers

At the lowest level of operation are the string level controllers. Transiting from standby mode to active mode of these string controllers could be triggered by the following three conditions:

The string controller observes a decrease in its local string level application utility.

The string receives a string failure event from RSS.

The string controller receives a wake-up call from its mission level controller.

For future implementations of the control system, we planned that the string controller will enter active mode from standby mode for other reasons not considered here for the sake of simplicity. For instance, it is feasible that the string controller may run a background job in standby mode to continually test if it can locally improve its application utility. If the string controller detects such a situation, then it will leave standby mode and act on this opportunity. There may also be other situations not yet considered where the string controller should enter active mode, but these will become apparent as more experience is gained with the system.

However, to address the first case for when the controller enters active mode, when a string level controller observes a drop in its application utility and the observed application utility is less than 90% of its average observed application utility, it enters its active mode. (The choice of a 90% threshold in this operation is an arbitrary starting point; a more informed choice for this value can be based on experience gained with the system. Given more in-depth system knowledge, it is also possible to use more advanced change detection methods drawn from the fields of estimation and detection theory, but this is an advancement on the current control system that will be addressed in future work.) If the string controller also receives a string failure event from the RSS, then a catastrophic hardware failure may have occurred which cannot be compensated for by the string controller. To verify this situation, the string manager attempts to restart the string (in place), and only if unsuccessful, the string controller relays the string failure event to the mission controller. The string controller then reenters standby mode because there is nothing further the string controller could do to compensate for the hardware failure.

If the string controller's observed application utility is less than 90% of the average observed application utility, but no string failure event has been received, then the decrease in the string's application utility is most likely due to resource contention. In this case, the string controller attempts to relieve its resource contention by decreasing the quality and throughput of the string. The controller does this by incrementally decreasing both its quality and throughput by 10% every sample period if possible. (Recall that above the sample period was arbitrarily chosen to be 1 second.) Although not stated earlier, we assumed that 1 second sample period is the minimum amount of time for the actions of the string controller on one sample period to be seen on the next sample period. If not, then the sample period should be adjusted accordingly.

On every sample period that the string's application utility increases due to the decrease of quality and throughput, the quality and throughput should be decreased again by 10% on the next sample period. On the first sample period that the string controller sees the application utility decrease, it stops decreasing quality and throughput, reenters standby mode, reinitializes its running average of observed local application utility and if the observed application utility due to the string controller tuning is not at least 90% of the original application utility mean, the string controller signals its mission controller that it was unable to sufficiently tune the string.

The overall heuristic of the string level controller in the case of resource contention is that the controller observes that its string needs to improve its application utility. The string controller tries to incrementally decrease its resource usage via actuations in the string's quality and throughput. The string controller continues to decrease the quality and throughput until those actions have an adverse effect on the string's application utility. When this occurs, the string controller signals the mission controller that it can do no more and reenters standby mode. Although we have outlined one possible method for the tuning of string-level application utility, other, more advanced methods are also possible. These control systems will be implemented in future versions of this work.

The wake-up call from the mission level can in general have three possible requests: deploying the string, removing the string, or adjusting the application utility threshold. Handling deploying or removing strings should be straight forward, therefore we'll only discuss the third case here. Upon receiving a wake-up call from the mission level controller to adjust the string's application utility threshold, it enters active mode and set the new threshold. If the current string application utility is below this new threshold, then it attempts to tune the string to pass the new threshold as described in the application utility dropping case; otherwise, the controller will return to standby mode.

Mission Level Controllers

Now that the behavior of the string-level controllers has been discussed in greater detail, we present our plan for how to implement the mission level controllers. Like the string level controllers, the mission level controllers normally operate in standby mode until the controller enters active mode and begins to take actions. The mission level controllers could be triggered to enter active mode from their standby mode under five conditions:

1. The mission controller observes a decrease in its local mission level application utility.
2. The mission controller receives a string failure event from one of its strings.
3. The mission controller receives a signal from a string controller that it was unable to sufficiently tune its string.
4. The mission controller receives a signal from the RSS that a hardware failure has occurred that impacts its strings.
5. The mission controller receives a wake-up call from the system level controller.

For future implementations of the control system, we planned on possibly allowing the mission controller to enter active mode from standby mode for other reasons not considered here. For instance, it is feasible that the mission controller may run a background job in standby mode to continually test if it can locally improve its application utility as was done with the string controller. As experience is gained with the resource allocation system, additional cases for the mission to take action may become apparent.

Once a mission level controller has entered active mode, there are three actions it can take: 1)it can send a wake-up call to the controllers of its strings in order to get those lower level controllers to perform their self-tuning operations, 2) the mission level controller can strings it can attempt to redeploy substrings of the strings under its control, and 3) the mission controller can send a signal to the system controller when it is unable to sufficiently raise its application utility. For operational simplicity the mission controllers are restricted so that each mission controller can attempt to redeploy portions of only one string at a time. Therefore, if a mission controller has multiple strings to adjust, the controller maintains an internal FIFO queue to order the strings it operates on. Due to the hierarchical nature of the control system, it is possible for the mission controller to have multiple strings perform their self-tuning simultaneously. However, it is a non-trivial task to design a mission level controller that would be able to perform multiple string adjustments simultaneously and this situation is not considered here. This issue will be addressed in later editions of this work.

It is now described what actions the mission level controller should take in each case of the five scenarios discussed above which would prompt the controller to enter its active mode:

If the mission level controller transitions to active mode because it receives a string failure notification from a string controller, then the mission controller attempts to adjust the string by redeploying parts of the string that have failed. If there are multiple failed strings, then the mission controller queues the strings in its FIFO queue and adjusts one string at a time until there are no strings left to adjust. When first attempting to adjust the deployment of a string, the mission controller first determines which substring in the deployed string is causing the string failure in order to identify which substrings of the string need to be redeployed. The mission controller would do this by querying the RSS in order to detect full or partial hardware failures along the string. The mission controller would also query the bandwidth broker and the string's substring managers to identify which substrings might be suffering from resource contention.

Once the substrings have been identified which are causing the string failure, the mission controller asks the pool managers to redeploy these identified substrings to different nodes in the pool. The goal of this strategy is that adjustments performed by the mission controller should be as small in scope as possible.

If the failed substrings cannot be redeployed to different nodes in the same pool, the mission controller needs to find other pools to place these controllers on. The mission controller, by querying the IA (which would use a resource allocation algorithm such as incremental mapping or bin-packing) for a list of possible redeployments of the strings.

Currently, the IA does not have the ability to generate lists of possible redeployment of substrings. This capability would have been needed to be added to the IA in order to implement this control system. We propose that if this modification were made to the IA, then the IA would return 3 possible deployments, if this many deployments are feasible. (Note that the value of 3 is arbitrarily chosen, but a better value will become evident as experience is gained in operating with the system.) Once the mission controller has its list of 3 possible string redeployments, the mission controller can use the utility estimation function in Equation 11 by querying the bandwidth broker and appropriate pool managers to identify the control action which would maximize the redeployed string's estimated utility. The utility estimation function proposed above can be calculated easily, so the main challenge in the string adjustment is the problem of generating the list of feasible control actions which may be very large if multiple pool substrings need to be moved.

One heuristic algorithm that could be used to generate its list of 3 possible partial string redeployments is to start by identifying, through queries of the RSS, the substrings of the string that are causing the greatest degradation in the string's application utility. Then, for each of these substrings, a candidate redeployment is generated where the substrings causing the degradation are moved to a different, randomly selected pool. The IA is then queried to test if this new redeployment is feasible. If the redeployment is feasible, then the redeployment is added to the list of the 3 candidate redeployments. This randomized search is continued until 3 feasible candidate redeployments are obtained. We planned that as more experienced is gained with the system, we would be able to develop a more intelligent redeployment search algorithm to generate a list of candidate redeployments.

Once the string has been successfully redeployed, the mission controller verifies that the observed application utility of the redeployed string is at least 90% of the mean application utility of the previous string configuration by making real-time observations of the string's behavior. If the new string does not meet the 90% threshold, then the mission controller sends a request to system controller that the system controller perform a redeployment of the insufficient strings. Note that the system level string redeployment is not taken as a first step in string recovery operations because the system-wide redeployment is global in scope and the underlying philosophy of the hierarchical control system design is that fast, local actions are generally preferable to slower, often more expensive global actions, except in designated cases such as command directives coming from the user.

If the mission controller receives a signal from the RSS that a hardware failure has occurred that impacts its strings, the mission controller first identifies which of its strings are impacted by the hardware failure. Then the mission controller attempts to adjust these strings through partial deployments as was done in the case of a string failure.

Similarly, if the mission controller receives a signal from a string controller that the string controller was unable to sufficiently tune the string, then the mission controller attempts to adjust the string as done above with failed strings. Also, if the mission controller observes a drop in its application utility below the 90% threshold, the mission controller identifies which of its strings are operating in a sub-optimal manner. The mission controller then sends wake-up calls to the controllers for those strings to perform their self-tuning operations.

Upon receiving a wake-up call from the system controller, the mission controller first queries its string controllers to see if any string failures have occurred. If string failures have occurred, the mission controller commences string recovery operations for these failed strings as described earlier in this subsection. If no string failures have occurred, the mission controller sends wake-up signals to all of its strings to commence their local self-tuning operations.

System Level Controllers

Now that the behavior of the planned string and mission level controllers has been discussed in greater detail, we present our initial plan of the system level controller. Like the lower level controllers, the system controller normally operates in standby mode until a transition to active mode after which the controller begins to take actions. The system controller could be triggered to enter active mode due to four conditions:

The system controller observes a decrease in the system level application utility.

The system controller receives a request from a mission controller to redeploy or deploy a subset or all of its strings.

The system controller observes a catastrophic hardware failure via RSS with system-wide implications (i.e., multiple mission failures).

The system controller receives user commands.

Once the system controller has entered active mode, there are three actions it can take. It can send a wake-up call to (one or more) mission controllers to get them to attempt to improve their local Application Utilities specific to their missions; it can attempt to reallocate resources for strings in a mission or multiple missions; and it can perform full reallocations of system resources.

When the system controller receives a request from a mission controller to redeploy or deploy a subset or all of its strings, the system controller initially attempts to generate a partial redeployment via the IA for the failed mission strings while leaving the resources of all other missions intact. (This incremental deployment functionality is not currently developed in the IA algorithm suite.) If there is no feasible partial redeployment for the failed mission, then the system controller performs a system-wide resource reallocation and reinitialization that could potentially impact other missions in the system

Upon the observation of a catastrophic hardware failure via RSS with system-wide implications, the system controller then attempts to redeploy all strings affected by the hardware failure as described above. There are some hardware failures, such as pool failures that may be best handled at the mission level when the pool failure affects the operation of only one mission. For the control implementation in this paper, a hardware failure is determined to have system-wide implications if the failure affects more than one mission. However, this heuristic threshold for when the system controller should take action in this case will be refined as more experience is gained with the system.

The system controller also enters its active mode if it receives external user commands to perform various actions. The way these requests are handle is very much the same as in the mission level controller. Instead of dealing with strings as in the mission level controllers, at the system level we are dealing with missions. For instance, if the ARMS user wants to deploy an extra mission in the system, then the system controller would send the allocation request for the mission to the IA and then reenter standby mode. Similar operations would occur if the external system user desires to terminate a mission or adjust the application utility threshold from 90% to 95%

5.2.2.3 Pool Failure Example

As an example of how the control system and information acquisition operations described above could operate, consider a system with three pools and multiple missions deployed and operating normally. In this example, we describe the operation of the control system and highlight how information flow occurs to and from various controllers.

If one pool were to fail, several events occur and are reported to controllers at various levels. Namely, a string level controller for a string with substrings deployed on the pool would observe a sharp decrease in the string's observed application utility and receive the string failure event. The mission level controller would then obtain information about the drop in the string's application utility from RSS, the report of string failure from the string controller, and the pool failure from RSS. The system level controller would also observe a drop in system application utility from the RSS, possible mission failures reported from mission controllers, and the pool failure event from RSS. (Note that individual string failures due to a mission failure are not reported to the system controller.) We discuss how these observations and information are passed in the control hierarchy and how decisions are made in our proposed multi-layer controller implementation

In our example, the string controller will observe two events: a string application utility drop and a string failure. . For simplicity, we assume that string controller will first attempt to restart the string with the current deployment before taking other actions. As discussed in Section 5.2.1, a more robust mechanism needs to be established to handle the different arrival orders and times of these events. In this case, the string controller would attempt to handle the string failure event but not the utility drop event. This is because string tuning after string failure would not result in any improvement of the observed application utility. However, since the string controller cannot accommodate the overriding string failure event locally, it would pass this event to its mission controller.

The mission controller also receives information about its drop in mission level application utility and would receive information about pool failure from the RSS. In this example it is assumed that the pool failure involves multiple missions, so system level coordination would preempt the mission controller.

The system level controller, in addition to receiving pool failure events, would observe the drop in the system level application utility. After observing these occurrences, the system controller determines that a system-wide reconfiguration can be performed by querying the IA.. At this point, the initially planned system controller attempts to redeploy the failed substrings originally deployed on the failed pool to surviving pools using the system level utility estimation function.

5.3 Dynamic Resource Management Simulation and Algorithm Refinement

We now describe how we refined our initial plan for DRM in ARMS. We took a bottom-up approach for our design refinement. We first refined our string-level resource controller. We then refined our mission-level resource controller and then our system resource controller which handled multi-mission coordination.

Our refinement of the resource controllers progressed hand-in-hand with the aid of hierarchical Matlab/Simulink models of the ARMS resource system. We developed resource models at each layer of the string-mission-system hierarchy to simulation the operation of our resource controllers. Based on our simulation results with the models, we were able to adjust our design to improve performance as measured by our utility measures and simplify our designs.

5.3.1 String Control

We started by developing a Simulink model for strings with a generic workload generator for simulating various computing jobs performed by the application sting with adjustable quality, throughput and real-time deadlines. Processing power and bandwidth can also be modified in the model to simulate dynamic properties of the strings' computing environments. We developed the simulation model for two simulation runs as a comparison.

During the first experimental run, a static resource control system was used that does not alter string job quality or throughput when changes in the rate of utility accumulation are observed on the feedback signal.

On the second experimental run, we deployed a simulated dynamic resource control system to control job quality that samples the change in utility accumulation at 15Hz. Note that 15Hz is the rate at which jobs are sent to be processed by the string, so the controller updates at the same rate that jobs are released to be processed by the string. Therefore, one control action can be taken for every job processed by the string.

For the dynamic string controller, on every sampling period when the average amount of change of the utility accumulated by the string per sampling time is negative, the dynamic resource controller cuts the quality of jobs in the string by half. On every sampling period when the change in utility accumulation per sampling time is positive, the dynamic resource controller increases the quality of jobs in the string by 20% as long as the resulting quality remains less than the initial allocation of 2 units. If the resulting quality is greater than the initial allocation of 2 units, then the resulting quality is set to 2 units so that the string does not process jobs with a higher quality than initially requested.

The string controller used on the second experimental run is a modification of the proposed string controller. We used this string controller because it will eventually return the string job quality to the initial allocation level if the faltering communication link recovers its lost bandwidth. The string controllers do not observe the occurrence of the communication link anomaly directly, but observe the utility accumulated by the string as jobs are processed.

5.3.1.1 String Comparison Setup

For our initial experiment, one application string is deployed over three pools and two communication links. The string is initially allocated sufficient bandwidth resources along the communication links to successfully process 15 jobs per second with a desired quality of 2 units and a hard deadline of 0.08 seconds. Jobs meeting the deadline have a reward of γ=0.1units per job and jobs not meeting the deadline have a penalty ρ=0.1units per job.

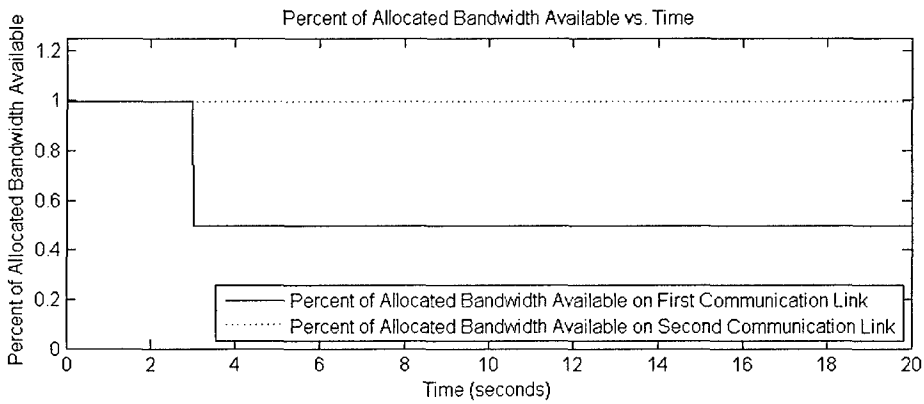

Figure 32: How the allocated bandwidth available to the string changes with time

The string application utility function is used to measure the performance of the string and as a feedback control signal to the string controller. We assumed for this experiment that the string does not suffer from any catastrophic failures and is therefore always available. Hence, the availability factor in the utility function (Equation 6) is set to a=1.

Two experimental runs were made during the experiment using our Matlab/Simulink string model. Both runs lasted 20 seconds. For both runs, a communication link anomaly is introduced at t=3 seconds that causes the first communication link to lose half its bandwidth. This change in available bandwidth causes sufficient delay in job processing such that with the initial allocation of resources, jobs processed by the link miss their deadlines. A graph showing how the allocated bandwidth available to the string changes with time can be seen in Figure 32.

5.3.1.2 String Control Simulation Results

In Figure 33 there are three subgraphs that show measurements taken from the first run of the experiment when a static resource allocation strategy was used. Note that the experiment is run from t=0sec to t=20sec and the bandwidth availability in the first link decreases by 50% at t=3sec.

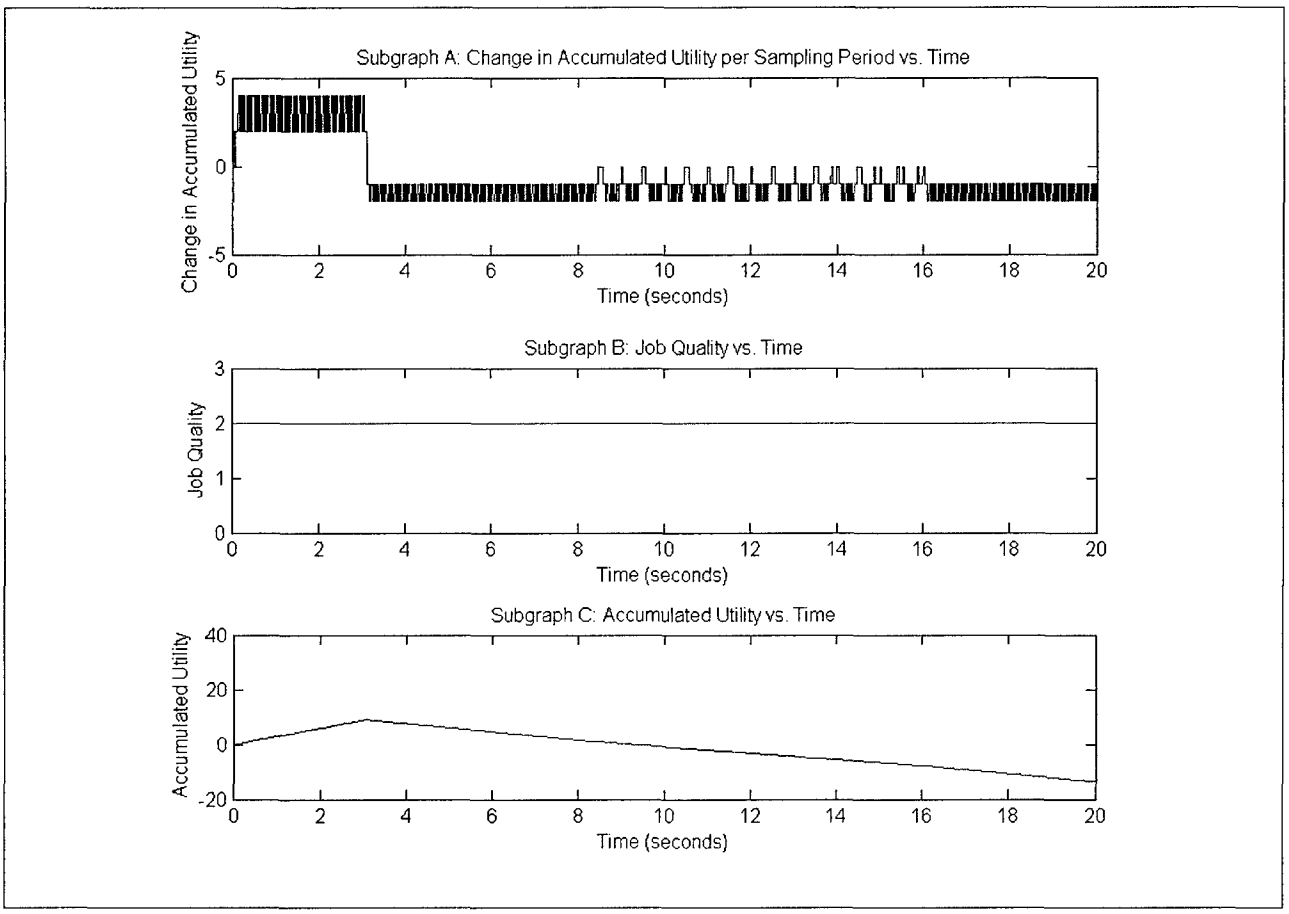

Figure 33: Experimental Results Due to Static Resource Strategy

Subgraph A in Figure 33 shows how the change in utility accumulation per sampling period changes with time. Prior to the link anomaly at t=3sec, the allocation of resources to the string allows all jobs meet their deadlines, so the change in accumulated utility per sample time is always positive before t=3sec. However, because of the link anomaly, after t=3sec, jobs are unable to meet their deadlines when job quality is 2 units. Therefore, all jobs miss their deadlines after t = 3sec and all jobs get assigned a penalty for missing their deadlines. Consequently, the change in utility per sample time is always negative after t=3sec.

Note that there is discontinuous "jitter" in the change in utility per sample period measurement during the entire run of the experiment. This is due to the effects of measuring the change in utility when rewards and penalties are accumulated at discrete instances in time. Furthermore, from approximately t = 8sec to t = 16sec, there are several spikes on the change in utility per sample period vs. time graph. These spikes are due to quantization effects associated with the Simulink simulation tool because it simulates the application string at discrete time steps. These spurious behaviors would disappear if a finer time step were used in the simulation and hence they can be disregarded.

Subgraph B in Figure 33 shows the quality of jobs being processed by the string over time during the first run of the experiment. The job quality is held constant at the initial setting of 2units because a static allocation is used.

Subgraph C in Figure 33 shows the accumulated application utility of the string over time. Up until t = 3sec, the utility of the string increases because all jobs sent to the string are processed before their deadlines. After t = 3sec, because of the decreased bandwidth available over the link, the accumulated application utility in Subgraph C of Figure 33 never increases which implies that all jobs miss their deadlines after the link anomaly.

In Figure 34 there are three subgraphs that show measurements taken from the second experimental run when the dynamic resource allocation strategy outlined above was used.

Subgraph A in Figure 34 shows how the change in utility accumulation per sampling period changes with time. Note that there is "jitter" in the change in utility measurement in Figure 34 due to quantization effects.

Prior to the link anomaly at t=3sec during the second run of the experiment, all jobs meet their deadlines, so the change in accumulated utility per sample time is always positive. However, immediately after the link anomaly at t=3sec, jobs are no longer able to meet their deadlines and penalties are assessed for every job that misses its deadline. Therefore, the change in utility per sample time is negative immediately after t=3sec.

As can be seen in Subgraph B in Figure 34, the quality setting of jobs being processed by the string over time during the second run of the experiment is constant at 2units until the simulated effects of the link anomaly are observed on the sampling period immediately after t=3sec. Then, because the change in utility accumulation per sampling period is negative in the sampling period immediately after t=3sec, the string controller decreases job quality by 50%.

On the sampling period after the job quality is decreased, no jobs fail to meet their deadlines and the observed utility increases as observed in Subgraph C in Figure 34. Therefore, using the control logic outlined above, the controller increases job quality by 20%. On the next sampling period, no jobs fail to meet their deadlines and the simulated observed utility again increases. The controller then again increases job quality by 20%.

This process of checking the change in accumulated utility and incrementally increasing the quality setting continues until the controller raises the job quality to 2units, the initial allocation level, and jobs are again dropped due to the decrease in bandwidth. As can be seen in Subgraphs A and B in Figure 34, the controller again decreases quality by 50% and then incrementally increases job quality as dictated by the control logic and the observed changes in accumulated utility per time interval. The process of the controller dropping the job quality by 50% on the observation of a drop in utility accumulation and then incrementally increasing quality until the next observation of a drop in utility accumulation continues indefinitely.

As can be seen from Subgraph C in Figure 34, the accumulated utility of the string increases steadily until the link anomaly at t=3sec. Then there is a small dip in utility when a job is dropped, but after the one sample period, the accumulated utility continues to increase at a slightly slower rate after t=3sec due to the quality control logic in the string controller.

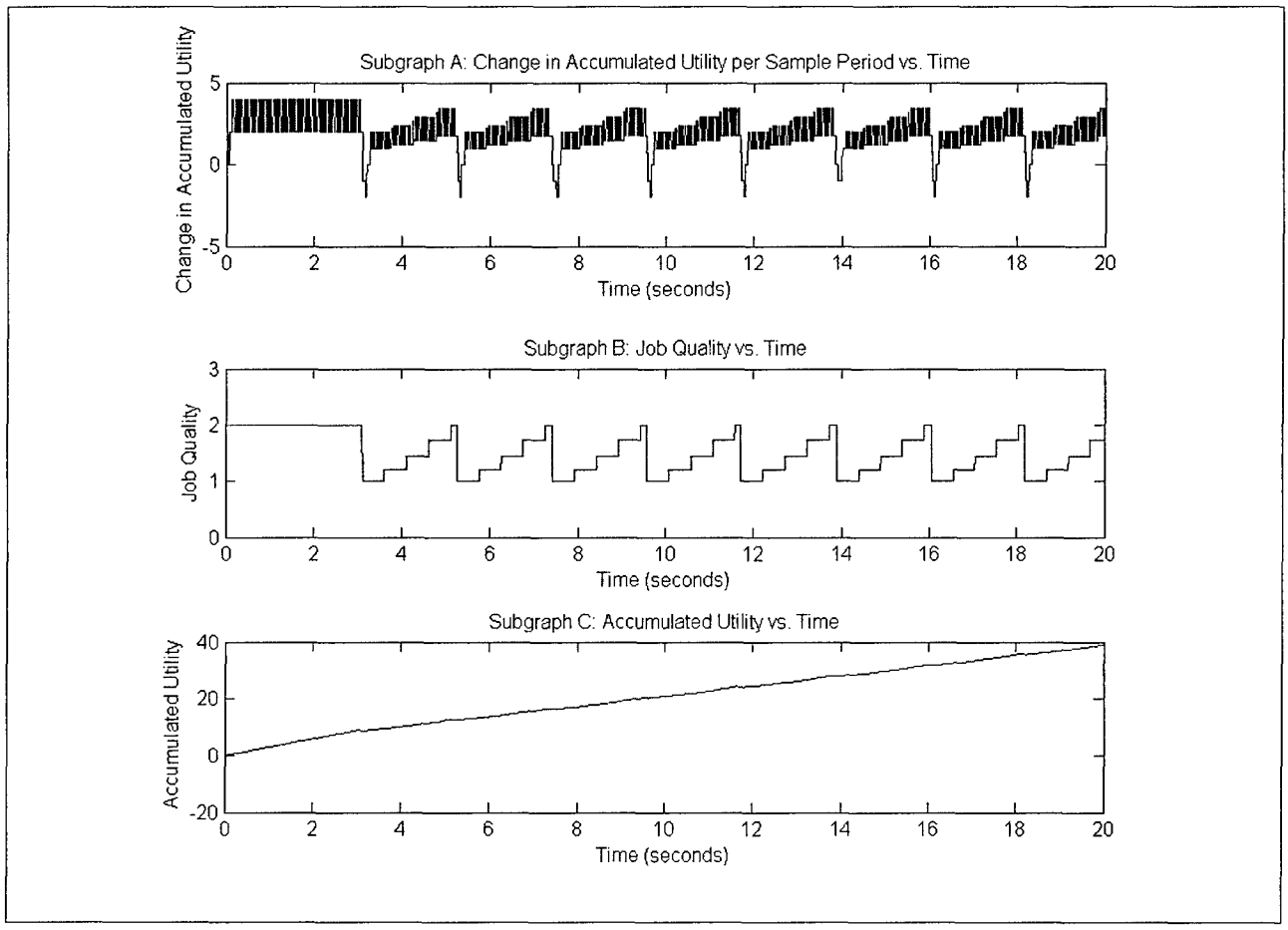

Figure 34: Experimental Results Due to Dynamic Resource Strategy

Note that a job periodically misses its deadline after t=3sec when the string controller attempts to bring job quality up to the initial level. Therefore, this control strategy is preferable only when the cost of missing additional job deadlines after initial failure is non-catastrophic or at least not too costly. If the cost of a job missing a deadline is very high, the control logic could be easily changed so that the controller does not attempt to increase quality when there is a chance of a job missing its deadline. However, although it is not demonstrated in this experiment, if the full bandwidth is ever restored to the communication link, the string control logic used in this experiment will eventually restore the string's job quality to its initial level of 2 units.

5.3.1.3 Discussion

This experiment demonstrated how using the application utility functions in conjunction with string control logic, a high level of string utility can be expected to be maintained to alleviate the effects of string anomalies that induce delays in job processing. During the first run of the experiment, all jobs missed their deadlines after the introduction of the anomaly due to link delays, but in the second run of the experiment when a string control system was used, almost all jobs met their deadlines due to the resource usage tunings of the string controller. This simulation provides more convincing evidence that the dynamic alteration of system resource usage can have a dramatic positive effect on the ability of a system to perform its desired tasks. This is illustrated in Figure 35 that graphs the simulated accumulated utility achieved using both resource usage strategies during the experimental runs.

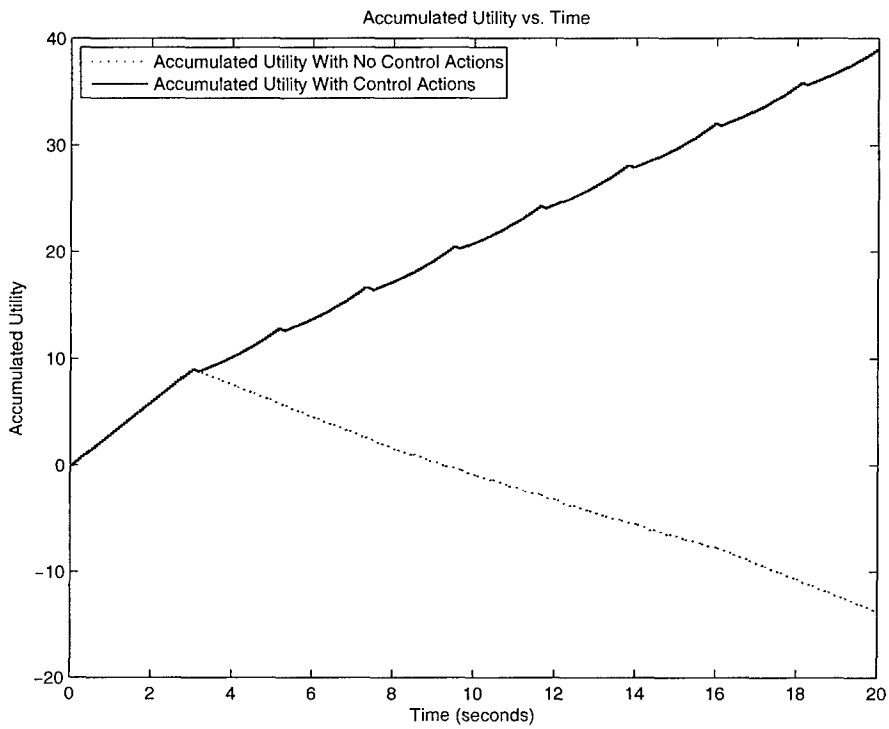

Figure 35: Comparison of Simulated Accumulated Utility.

5.3.2 First Approach to Mission Control

After refining the design of the string resource controller, we turned our attention to the mission controller. Strings are assigned importance values, and for GT4 Metric 1, warfighter value is computed such that it is better to deploy a higher valued string rather than any combination of lower valued strings. Therefore, for simplicity in our design in our first refinement of the mission control system, we started to refine the high-level control logic to select strings to deploy or kill in order to maintain a high warfighter value with respect to Metric 1. This mission controller is designed to maintain warfighter value in response to the simultaneous occurrence of system failures and user-driven changes in the warfighter values of its strings.

The goal of the mission controllers is to select strings to be deployed and to find resources which those strings can use. Therefore, we split the mission level control logic into two layers. The high-level mission control logic attempts to deploy strings by requesting string deployment plan from a low-level mission control logic. After receiving a deployment plan request, the low-level control logic attempts to find resources that would be sufficient for the requested string to use to run. If the low-level control logic finds a deployment plan for the requested string, then this information is returned to the high-level logic. If the low-level control logic cannot find a deployment plan for the requested string, then this information is also returned to the high-level logic. When the high-level control logic is given a deployment plan for a string it wants to deploy, the high-level control logic deploys the string by sending the appropriate information about the string's substrings to the Pool Managers of the pool where the string are placed.

The low-level controller always attempts to find deployments for strings to keep all applications associated with the string on the same pool. If it is not possible to keep all of the string's application on the same pool, then the low-level logic tries to find a resource deployment for the string that would minimize the amount of inter-pool communication bandwidth required by the string. Similar to utility-driven high-level control logic, the low-level control logic uses a resource utility measure when finding resources for strings in order to ensure load-balancing. The functionality of the low-level control logic is very similar to the previous functionality of the IA.

5.3.2.1 High-Level Control Logic

The high-level controller is primarily a reactive system in that it responds to information about failures and changes in the warfighter values of strings. It is designed such that if the mission receives information about a failure that affects its deployed strings or information about a revaluation of its deployed strings, then the mission controller kills and restarts the operations of its high-level logic. In this manner, the mission controller can respond to multiple events that occur before the mission controller can complete already started operations. The high-level control logic attempts to deploy strings at system initialization, recover warfighter value in response to failures and redeploy strings in response to changes in strings' revaluation with respect to warfighter value. Consequently, the high-level control logic needs access to information about its strings' warfighter value and information about which of its strings are operating.

The high-level mission control logic receives information about failures from the RSS, and information about string revaluation from the user via the system controller. Because the low-level control logic uses functionality associated and information already associated with the IA, we place the low-level control logic on the IA. This modified functional layout of the MLRM can be seen in Figure 36.

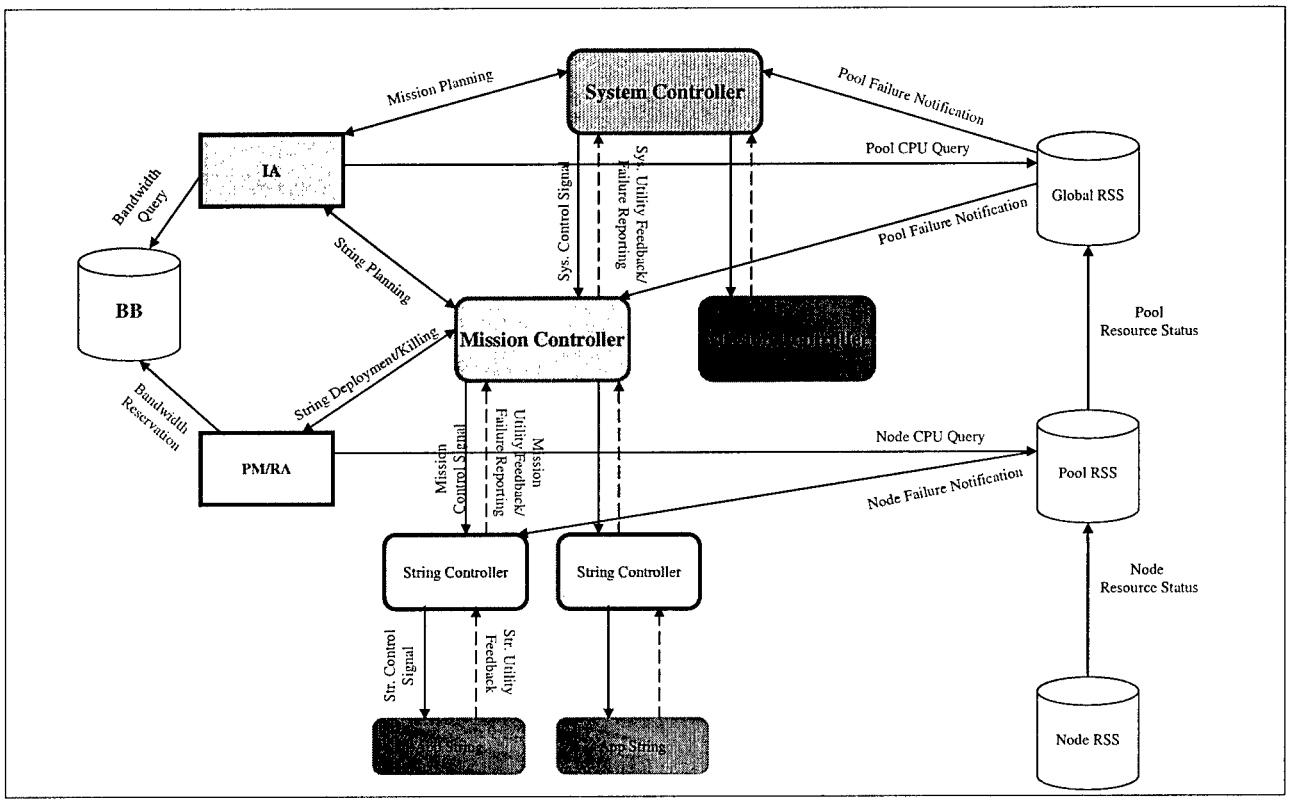

Figure 36: MLRM Components and Interaction

As an informal description of the operation of this initial refinement of the high-level mission control logic of the mission controller, when the controller receives information about a pool-level failure or a change in string valuation, the high-level logic generates a list of the mission's strings which are not deployed but whose redeployment should be attempted. The high-level sequentially attempts to deploy the strings in this list. If any strings are killed by the mission controller to free up resources for higher value strings, then these killed strings are added to the list of strings for which deployment should be attempted. Strings are removed from the list of strings to attempt deployment for two reasons. Most basically, strings are removed from this list when they are deployed. Strings are also removed from this list when they cannot be deployed and there are no deployed lower-valued strings which can be killed to free up additional resources for the string.

The high-level logic iteratively attempts to always deploy the highest value string in its list of string for which deployment should be attempted. When the high-level logic attempts to deploy a string, the high-level logic requests that low-level logic find a deployment plan for the most important string in the list of non-operational strings.

```
highLevelLogic
{
        D is the set of strings deployed.
        U is the set of strings whose deployment should be attempted.
        While U is non-empty
        {
                s = highestValueString(U);
                deploymentPlan = lowLevelLogic(s);
                deployable = (deploymentPlan == null);
                if deployable
                {
                        deploy(s,deploymentPlan);
                        D.add(s);
                        U.remove(s);
                }
                else
                {
                        d = lowestValueString(D)
                        if value(s) > value(d)
                        {
                                kill(d);
                                D.remove(d);
                                U.add(d);
                        }
                        else
                        {
                                U.remove(s);
                        }
                }
        }
}
```

Figure 37: High-Level Control Logic, First Attempt.

If the low-level logic successfully returns a deployment plan, then the high-level logic deploys that string using the returned plan and removes the string from its list of non-operating strings. If the low-level logic does not return a valid deployment plan, the high-level logic kills the mission's lowest value operating string if this string's importance value is less than the importance value of the string the controller is attempting to deploy (if there is such a lower-valued string). If a lower-valued string is killed, then this killed string is added to the list of strings which are not operating, but for which deployment should be attempted. If there are no such lower-valued operating strings to kill, then the original non-operating string is removed from the list of strings for which deployment should be attempted.

The high-level logic then iterates by again attempting the deployment of the highest value string on the deployment list. (Note that this next highest value string may be same as the last highest value string.) The algorithm ceases operation when its list of string for which deployment should be attempted is empty. In this manner, all strings which are not deployed, or are killed during the algorithms operation are always attempted to be (re)deployed.

If the mission controller ever receives updated information about failures or its strings' values while the algorithm is operating, then the algorithm is killed and restarted with the updated information. This algorithm is formalized in the pseudo code in Figure 37.

5.3.2.2 Low-Level Control Logic

The low-level control logic only operates when called by the high-level control logic. Because the low-level control logic attempts to find resources that the mission's strings can use, the low-level logic needs information about the requested strings' resource requirements. The low-level logic also needs information about the availability of pool-level resources in the system. Pool-level resources include information about the pools' ability to accept a proposed substring and the amount of free bandwidth on the inter-pool communication links. In order to maintain information about the ability of a pool to accept a substring, the low-level control logic may contain estimates of pools' resource slack. In order for the low-level logic needs to verify that a pool can accept a substring, the low-level logic should be able to query Pool Managers for this information.

The low-level control logic develops plans to place substrings on pools. Once a substring is assigned to a pool, the pool's Pool Manager assigns the string's substring to individual nodes in the pool. Therefore, the low level control logic does not need information about the availability of individual nodes in pools.

We have designed the low-level control logic to place the entire string onto the same pool is possible.

Strings are comprised of individual applications that process data and communicate the data to other applications in the pool. As was discussed in the introduction, inter-pool bandwidth is generally much more expensive than intra-pool bandwidth. Therefore, to find a deployment plan for a string that cannot fit on a single pool, our low-level control logic splits the string into two substrings s_1 and s_2 to minimize the bandwidth required to pass data between the substrings. Therefore when s_1 and s_2 are placed on different pools, they use as little inter-pool bandwidth as possible. Our resource utility measure is described in the next section in greater detail, but we use it as a measure of the efficiency of proposed resource allocations.

Once the strings s is split into the substrings s_1 and s_2, the low-level control logic is called recursively to also place these substrings on pools individually in the same manner as the original string. Therefore, when the low-level control logic is called recursively for s_1 and s_2, it is first attempted to place these substrings wholly onto a pool. If this is not possible, then the substrings will potentially be split again. (If the substrings can be split.) Note that if a (sub)string is comprised of exactly one application, then it cannot be further split into substrings. The low level control logic algorithm is formally presented in pseudo-code in Figure 38.

```
//low-level logic
lowLevelLogic(s)
{
        deploymentPlan = selectPool(s);
        if(deploymentPlan != null)
        {
                return deploymentPlan;
        }
        else if(num of application in s >= 2)
        {
                Split s into two substrings s1, s2 to minimize
                        interpool bandwidth.
                deploymentPlan1 = lowLevelLogic(s1);
                deploymentPlan2 = lowLevelLogic(s2);
                if deploymentPlan1 != null & deploymentPlan2 != null
                {
                        return (deploymentPlan1,deploymentPlan2);
                }
                else
                {
                        return null;
                }
        }
        else
        {
                return null
        }
}
```

Figure 38: Low-Level Control Logic, First Attempt.

If there exists multiple pools that can accept a (sub)string, then instead of selecting one of these pools randomly for the (sub)string's deployment plan, we design the low level control logic to select the optimum pool as measured by resource utility. This aspect of the (sub)string resource selection algorithm is seen in Figure 38. We discuss our resource utility measure in the immediately following section.

5.3.2.3 Resource Utility Computation

Our initial refinement of the mission-level control algorithms uses a resource utility measure to decide on where to place strings in the system. We have formulated a resource utility measure that when used in the low level control logic, leads the system to generate load balancing deployment plans. We felt that this method was advantageous because hardware failures will have less of a potentially catastrophic affect. That is any one hardware failure will be less likely to lead to an inordinate number of string failures.

To compute resource utility, suppose that over the pools $P_1, P_2, \ldots P_p$, there is a resource slack of $ps_1, ps_2, \ldots ps_p$. Also, over the inter-pool communication links $L_1, L_2, \ldots L_l$, suppose there are resource slacks of $ls_1, ls_2, \ldots ls_l$. As previously discussed[i], resource slack is a measure for the amount of free resources on the pool.

We define the pool resource utility to be $\sum_{i=1}^{p} \sqrt{ps_i}$ and the link resource utility to be $\sum_{i=1}^{l} \sqrt{ls_i}$. The total resource utility to be

$$RU = w_P \sum_{i=1}^{l} \sqrt{ps_i} + w_L \sum_{i=1}^{l} \sqrt{ls_i}.$$

This measure for resource utility implies that the highest utility is obtained by evenly distributing resource utility over the various pools and links. In the resource utility measure, the factors w_p and w_l are weighting factors. If it is much more important to conserve inter-pool bandwidth than pool resources, then w_l should be assigned much greater than w_p. The factors w_p and w_l will most likely need to be tuned when this control system is deployed on a real system.

```
selectPool(s)
{
        if(no pool can accept s)
                return null;
        else
        {
                Compute Resource Utility for all possible
                        pool assignments;
                p = pool associated with highest Resource Utility
                return (s,p);
        }
}
```

Figure 39: Resource Utility Driven Pool Selection Algorithm.

To illustrate the load balancing effect due to the usage of this resource utility measure, suppose $RU = w_P \sum_{i=1}^{l} \sqrt{ps_i} + w_L \sum_{i=1}^{l} \sqrt{ls_i}$, and a pool substring needs to be assigned to a pool that would diminish its pool's resource slack by s. If $ps_1 > ps_2$, then $\sqrt{ps_1 - s} + \sqrt{ps_2} > \sqrt{ps_1} + \sqrt{ps_2 - s}$. Hence there would be a higher utility of placing the string portion on Pool 1 instead of Pool 2.

5.3.2.4 Mission Simulations

To test our initial refinement of the mission control design, we also developed Matlab/Simulink simulations. We wanted to investigate the performance of the controllers with respect to GT4 warfighter Metric 1. Scenarios that we were particularly interested in three situations where a mission with multiple strings and the mission controller needs to respond to:

Hardware failures.

Changes in warfighter value that would drive the most important strings.

Changes in warfighter value while simultaneously recovering from hardware failures.

To evaluate our mission controller design scenarios, we simulated a system comprised of 5 pools with bounded inter-pool bandwidth. In the system there is one mission with 10 non-trivial, non-uniform strings. Warfighter Metric 1 was used to evaluate the warfighter value derived from the controller operating in these scenarios.

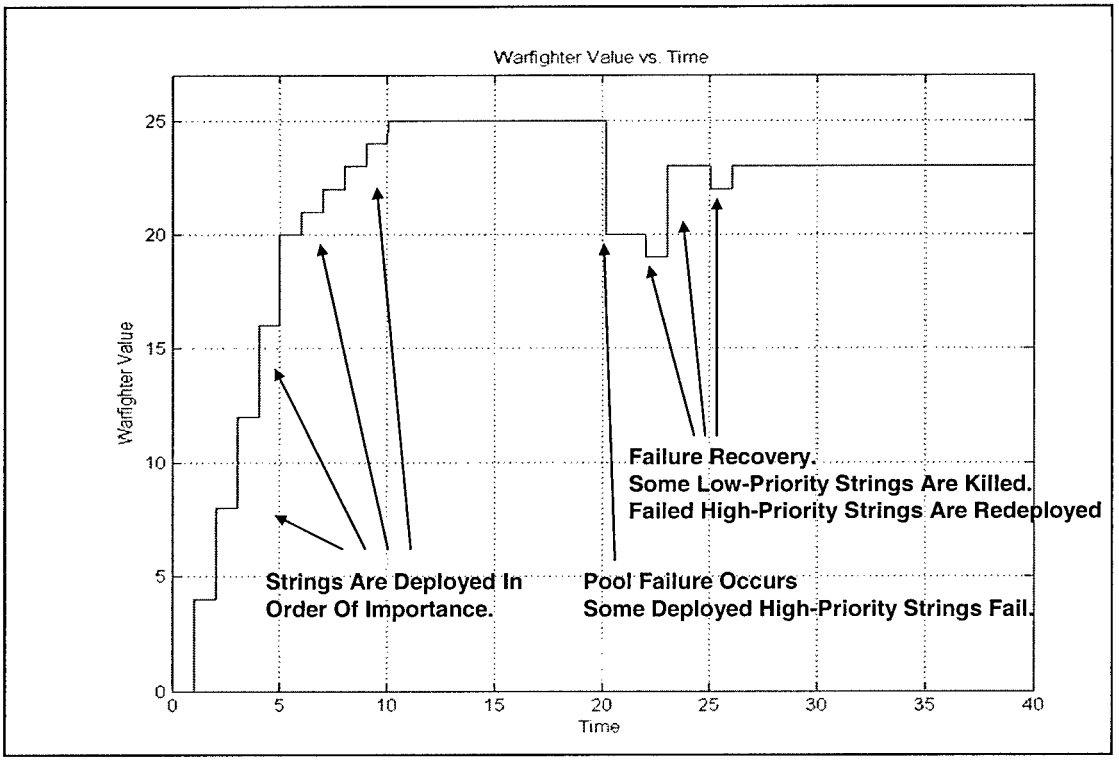

Figure 40: An overview of warfighter value evolution.

As a basic assumption to establish a baseline comparison, we assume that the mission controller only updates once a second. This may or may not be realistic for the ARMS system, but we needed to use a standard measure to evaluate the abilities of our control system.

Scenario 1

This scenario was used to test the performance of the mission controller with respect to hardware failures. For this scenario, a mission is initially deployed with 10 strings which are assigned a mixture of high and low values. There are 5 high importance strings with a warfighter value of 4: Strings 1, 2, 3, 4, 5, and there are 5 low importance strings with a warfighter value of 1: String 6, 7, 8, 9, 10.

After the mission is deployed, it needs to deploy its individual strings. One of the five initially operating pools is randomly selected to fail permanently at t=20sec. After this failure occurs, the mission controller should attempt to recover as much warfighter value as possible after the failure by either killing or (re)deploying its strings.

Graphs of warfighter value versus time for the Matlab/Simulink simulation of the mission controller in this scenario can be seen in Figure 40 and Figure 41. The graph in Figure 40 shows in broad strokes the operation of the system during this experiment, while Figure 41 shows the same data with more detailed information concerning the attempted deployments and killings of the mission's strings. As can be seen from these graphs, the mission's warfighter value drops immediately after the pool failure occurs, but by killing low priority strings, the mission controller is able to restart the failed higher priority strings.

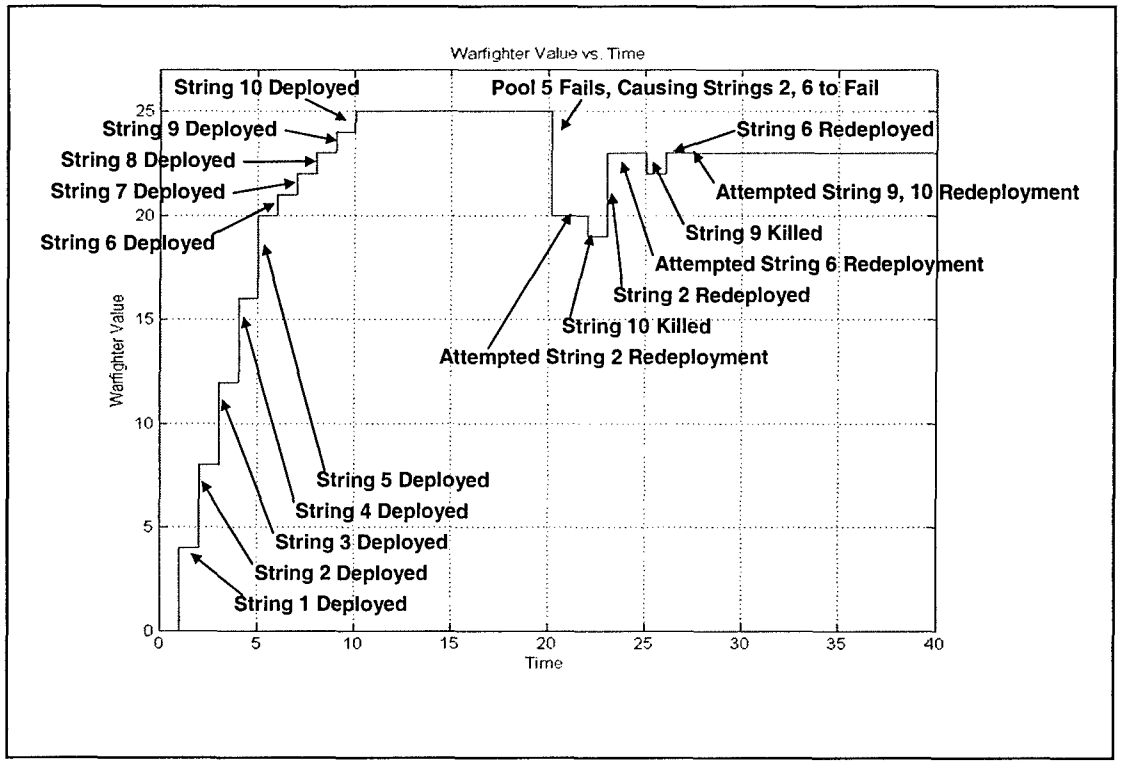

Figure 41: A detailed overview of warfighter value evolution.

In addition to the warfighter value graphs, Figure 42, Figure 43 and Figure 44 show snapshots about the details of the strings' deployments over time. Figure 42 shows how all of the system's strings are deployed after initialization, but before the pool failure occurs. Figure 43 shows how the mission's operating strings are deployed immediately after the pool failure, and Figure 44 shows how the mission's strings are deployed after the mission has attempted to recover from the pool failure.

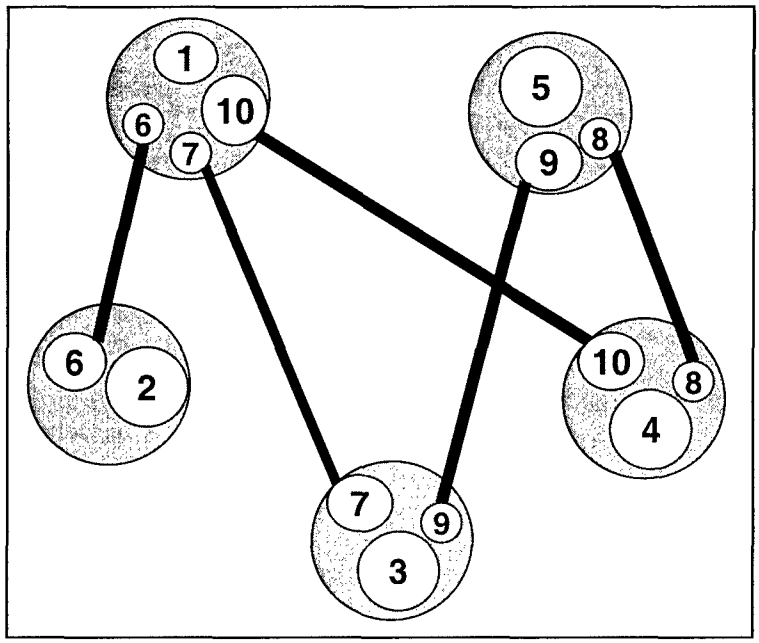

Figure 42: Scenario 1 String Configuration After Initialization

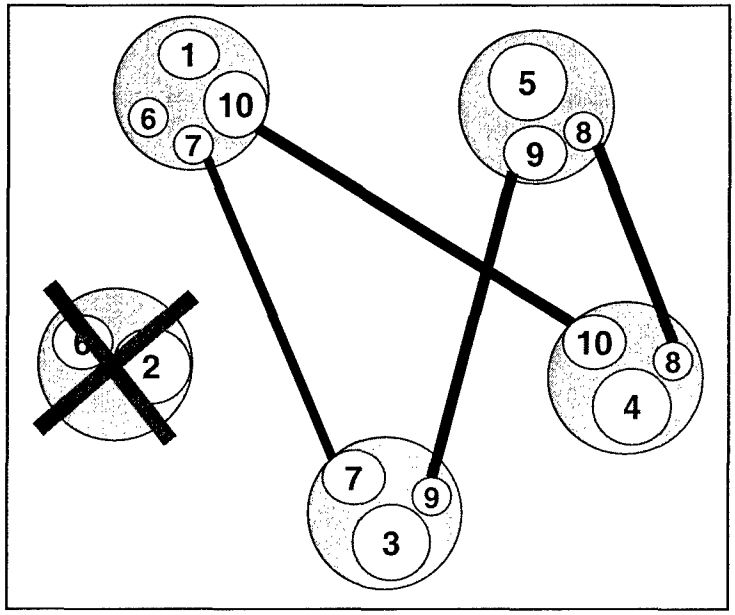

Figure 43: Scenario 1 String Configuration After Failure.

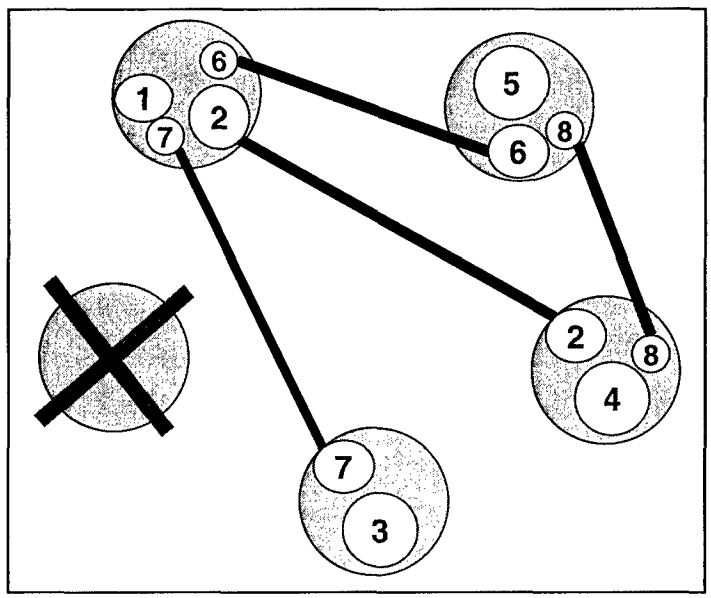

Figure 44: Scenario 1 String Configuration After Recovery

Scenario 2

This scenario is a continuation of the previous scenario where after the mission controller has finished recovering from the pool failure, the mission controller should respond to a change in string valuation. Therefore, at the beginning of this scenario, the system has one mission with 10 strings. Eight of the strings are initially deployed over 4 operating pools. There are 5 high importance strings with a warfighter value of 4: Strings 1, 2, 3, 4, 5, and there are 5 low importance strings with a warfighter value of 1: String 6, 7, 8, 9, 10. Strings 9 and 10 are initially not deployed.

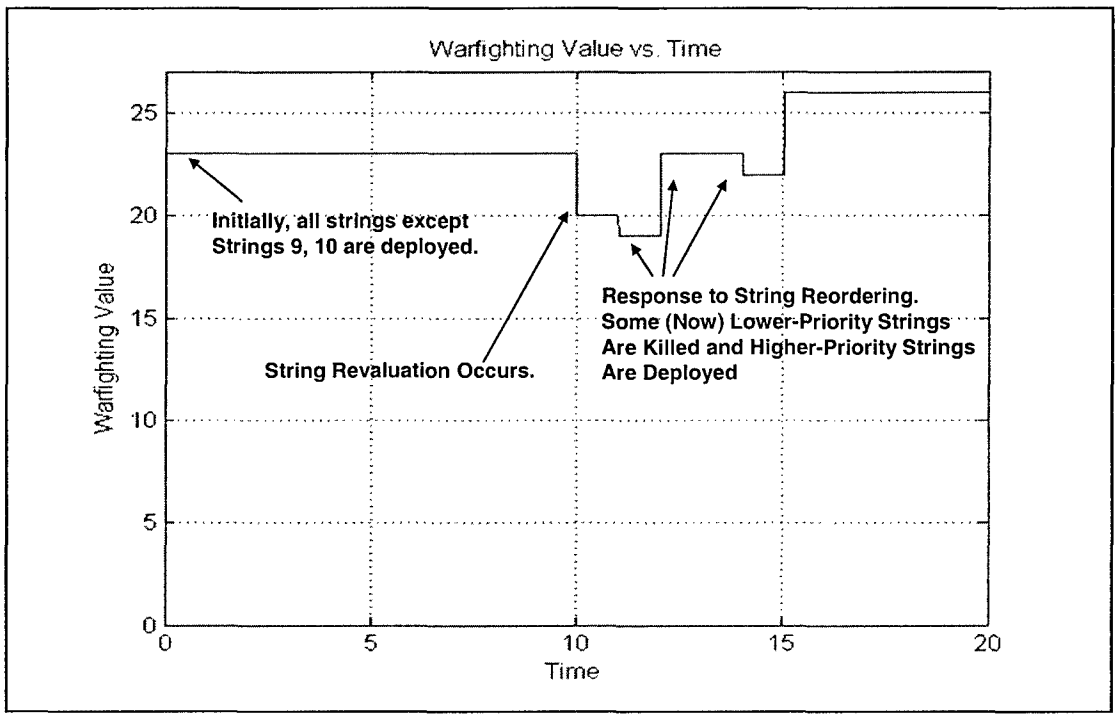

Figure 45: An overview of scenario 2 warfighter value evolution.

At t=10sec, the strings are revaluated such that after the revaluation there are 6 high importance strings with warfighter value of 4. These strings are Strings 1,2,3,4,9, 10. Additionally, after the revaluation there are 4 non-critical strings with warfighter value of 1 : String 5, 6, 7, 8. Note that after the revaluation, Strings 9 and 10 become high-importance, while String 5 becomes low importance. After the revaluation, the mission controller should deploy or kill its strings as necessary to obtain as high a warfighter value as possible.

Graphs of warfighter value versus time for the Matlab/Simulink simulation of the mission controller in this scenario can be seen in Figure 45 and Figure 46. The graph in Figure 45 shows in broad strokes the operation of the system during this experiment, while Figure 46 shows the same data with more detailed information concerning the attempted deployments and killings of the mission's strings. As can be seen from these graphs, the mission's warfighter value drops when the string revaluation occurs, but by killing low priority strings, the mission controller is able to deploy the newly high priority strings that were not previously deployed.

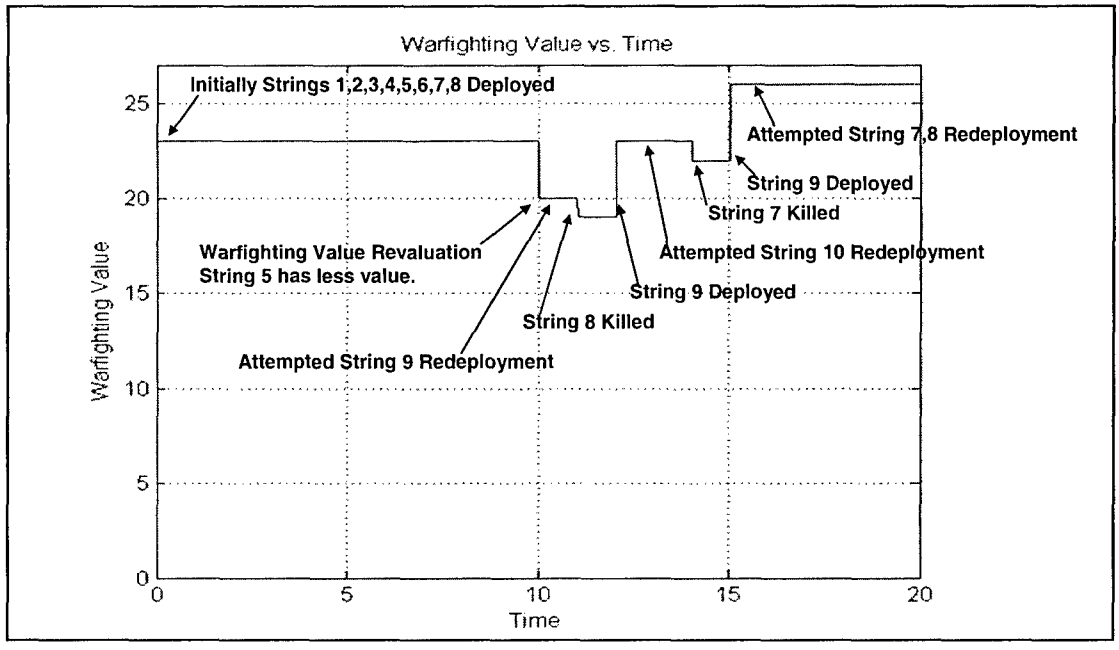

Figure 46: A detailed overview of Scenario 2 warfighter value evolution.

In addition to the warfighter value graphs, Figure 47 and Figure 48 show snapshots about the details of the strings' deployments over time. Figure 47 shows how all of the system's strings are deployed before the revaluation occurs. Figure 48 shows how the mission's operating strings are redeployed after the mission controller has responded to the string revaluation.

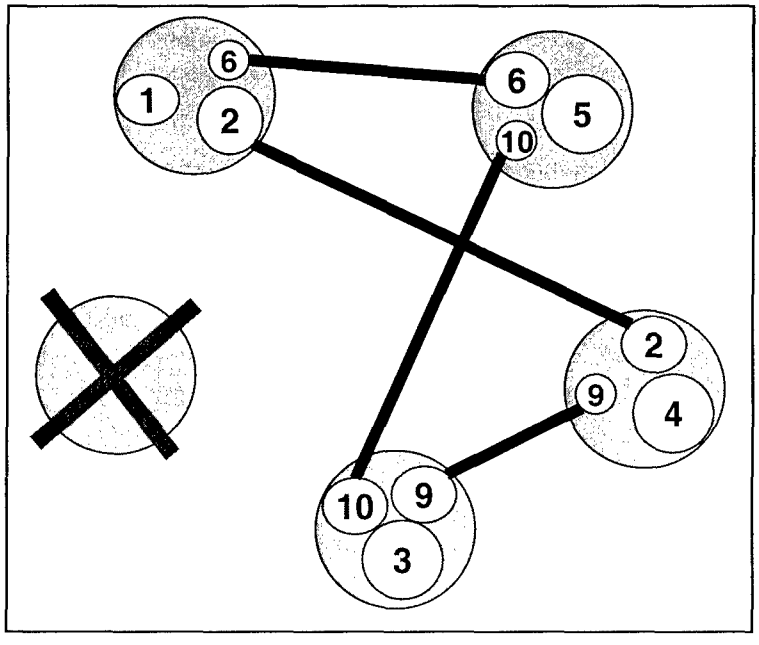

Figure 47: Scenario 2 String Configuration After Response to String Revaluation

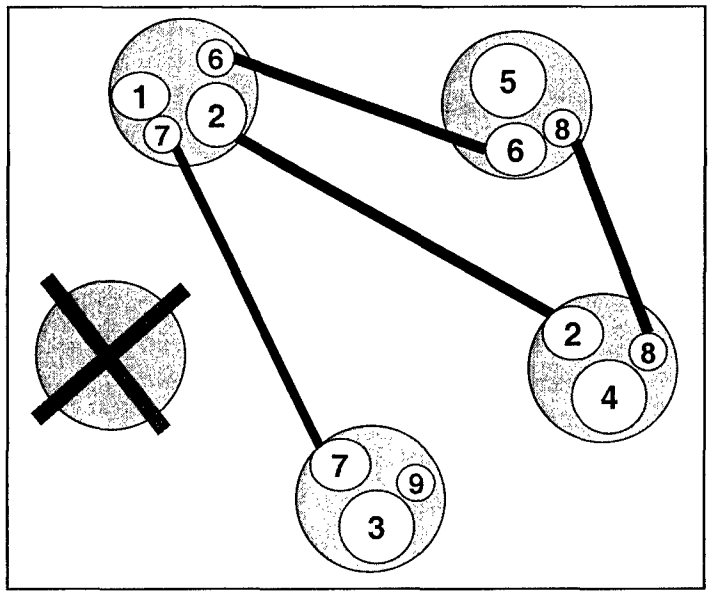

Figure 48: Scenario 2 String Configuration At T=0sec

Scenario 3

This scenario is a combination of scenarios 1 and 2. For this scenario, a mission is initially deployed with 10 strings which are assigned a mixture of high and low values. There are 5 high importance strings with a warfighter value of 4: Strings 1, 2, 3, 4, 5, and there are 5 low importance strings with a warfighter value of 1: String 6, 7, 8, 9, 10.

After the mission is deployed, it needs to deploy its individual strings. One of the five initially operating pools is randomly selected to fail permanently at t=20sec. After this failure occurs, the mission controller should attempt to recover as much warfighter value as possible after the failure by either killing or (re)deploying its strings. However at t=23sec, before the mission can completely recover from the pool failure, a revaluation of the mission's strings occur. The strings are revaluated such that after the revaluation there are 6 high importance strings with warfighter value of 4. These strings are Strings 1,2,3,4,9, 10. Additionally, after the revaluation there are 4 non-critical strings with warfighter value of 1 : String 5, 6, 7, 8. Note that after the revaluation, Strings 9 and 10 become high-importance, while String 5 becomes low importance. Because the revaluation occurs before the mission can completely recover from the pool failure, the mission controller needs to simultaneously respond to the effects of the pool failure and the string revaluation.

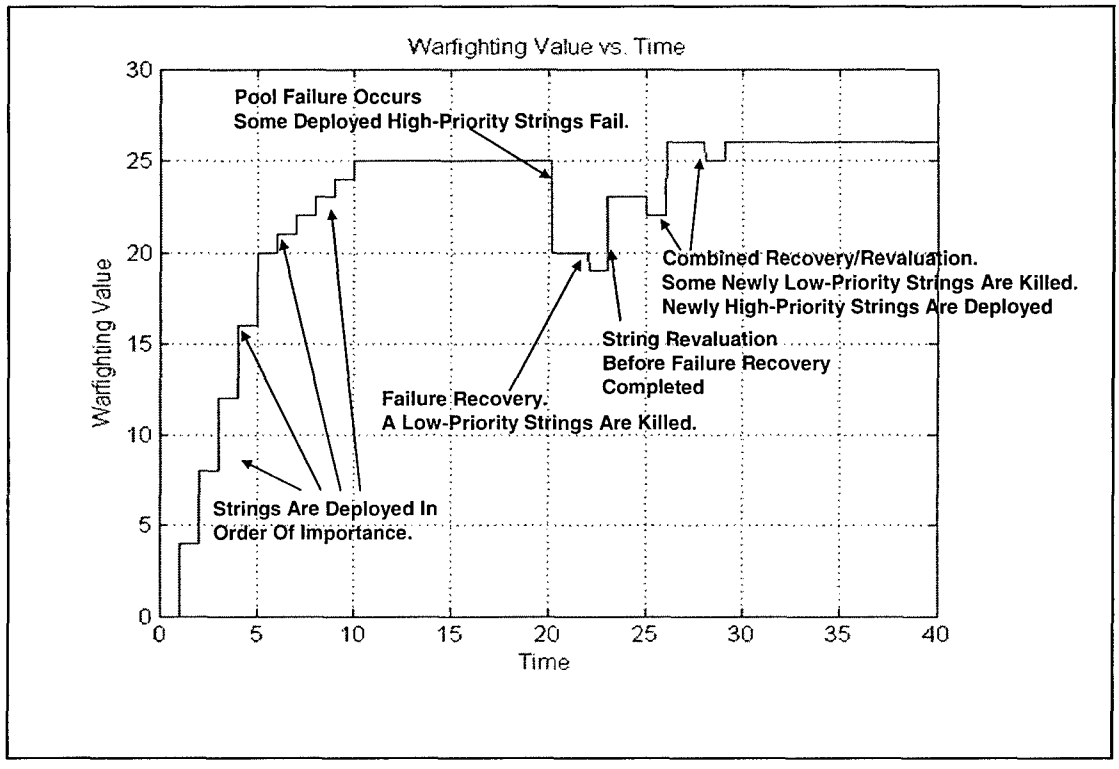

Figure 49: An overview of scenario 3 warfighter value evolution.

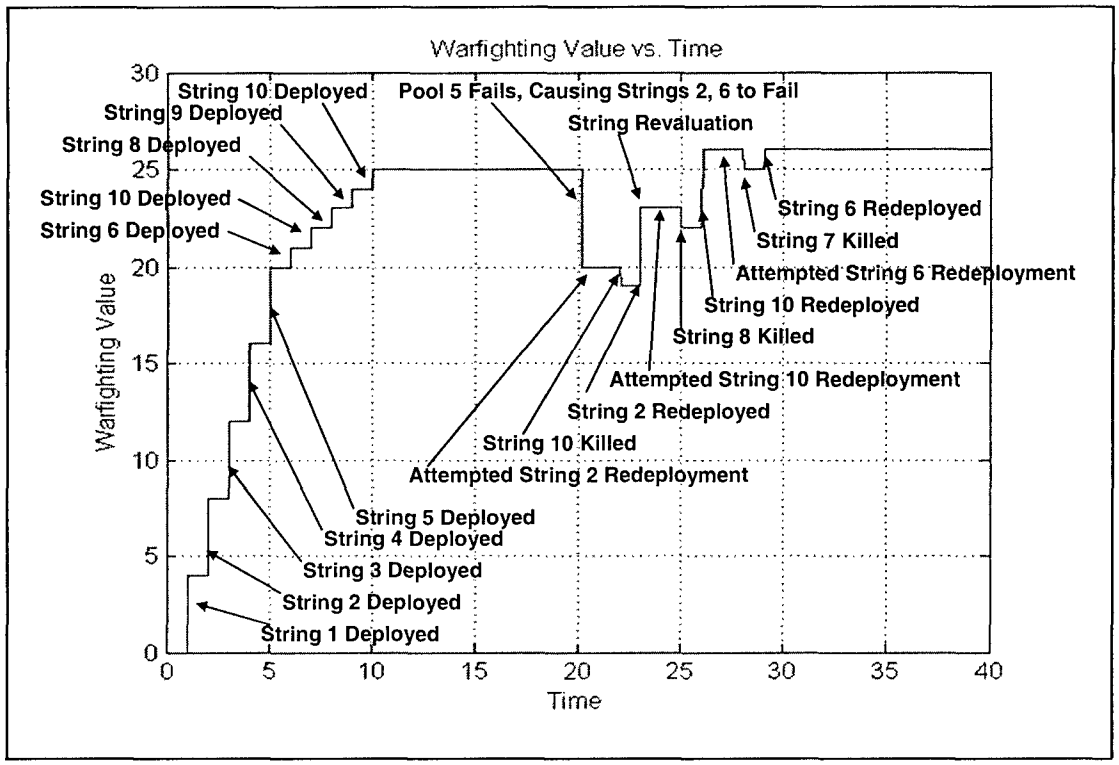

Figure 50: A detailed overview of scenario 3 warfighter value evolution.

Graphs of warfighter value versus time for the Matlab/Simulink simulation of the mission controller in this scenario can be seen in Figure 49 and Figure 50. The graph in Figure 49 shows in broad strokes the operation of the system during this experiment, while Figure 50 shows the same data with more detailed information concerning the attempted deployments and killings of the mission's strings. As can be seen from these graphs, the mission's warfighter value drops both when the pool failure occurs and when the string revaluation occurs. However, by killing low priority strings, the mission controller is able to deploy the newly high priority strings that were not previously deployed.

In addition to the warfighter value graphs, Figure 51, Figure 52, Figure 53 and Figure 54 show snapshots about the details of the strings' deployments over time. Figure 51 shows how all of the system's strings are deployed after initialization. Figure 52 shows the configuration of the system immediately after the pool failure has occurred and Figure 53 shows the configuration of the system when the string revaluation occurs. Figure 54 shows how the mission's operating strings are redeployed after the mission controller has finished responding to both the pool failure and the string revaluation.

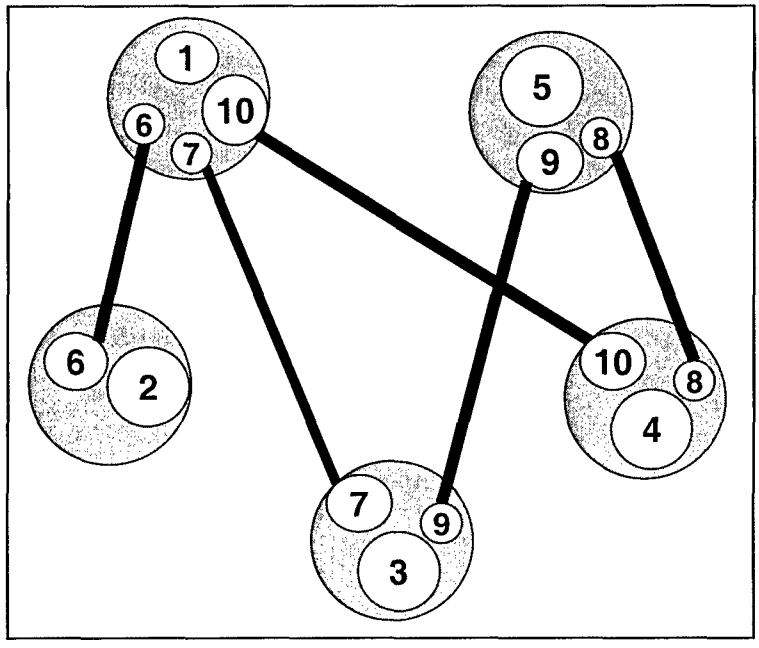

Figure 51: Scenario 3 String Configuration After Initialization

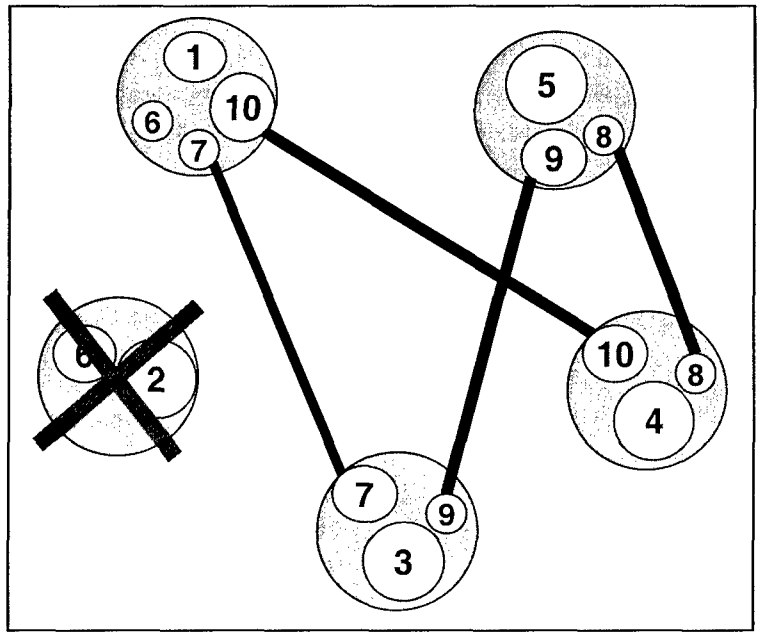

Figure 52: Scenario 3 String Configuration After Failure.

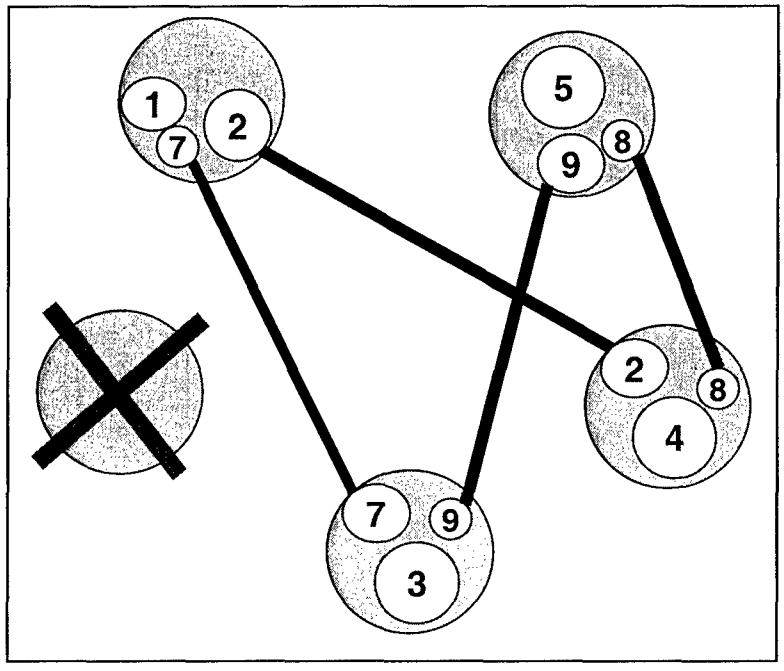

Figure 53: Scenario 3 String Configuration On String Revaluation

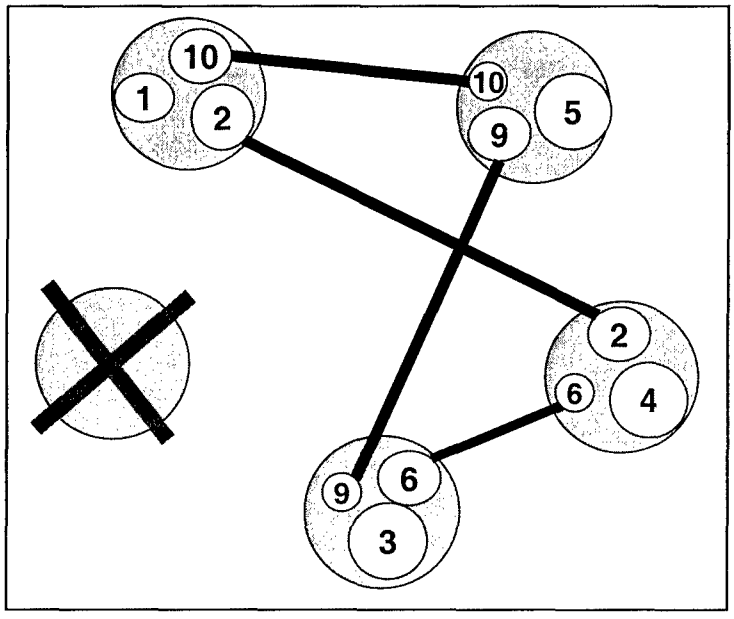

Figure 54: Scenario 3 String Configuration After Response to Failure and Revaluation

5.3.3 Second Approach to Mission Control

With our experience in mission control design from our first round of refinement and simulation, we next attempted to improve the mission controllers to account for both GT4 Warfighter Metrics and our Application Utility Measure. We refined the ARMS mission control system, which consists of three algorithmic components, to dynamically decide which strings to deploy/kill/redeploy and what resources the deployed strings should be directed to use. These algorithmic components take input from the user and interact with the external ARMS system and one another to perform their resource allocation operations. A high-level schematic of the interactions between the mission controller algorithm components can be seen in Figure 55.

In order to deploy the mission's strings, the mission controller is given input parameters from the user about which strings the mission should attempt to deploy, information about these strings' importance, and resource requirements. As the mission operates, the user may change any of these input parameters during run-time.

The sorting logic in the mission controller takes the inputs from the user and generates a sorted list of the mission strings using the sorting order specified by the user. This sorted list of strings is passed as input to the mission controller and can have a large impact on the performance of the system. The mission controller runs the sorting logic for all strings whenever the user inputs new information to the mission controller.

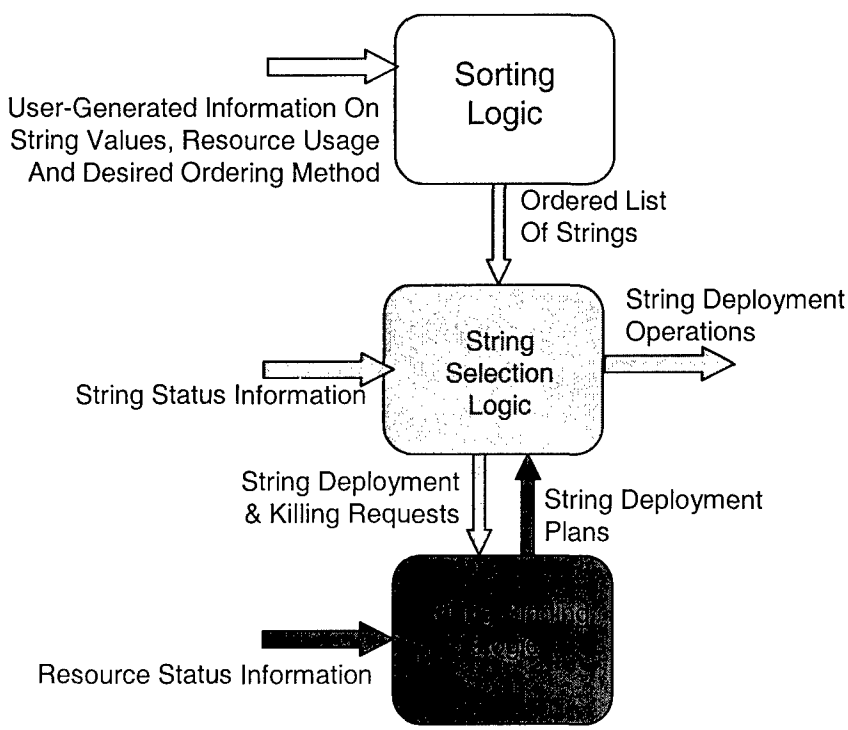

Figure 55: Mission Control Inter-Algorithm Information Flow

After the sorting logic has completed operation, it passes the sorted list of strings to the string selection logic. The string selection logic, which we previously called the high-level control logic, determines which strings to deploy/kill/redeploy based on the strings' locations in the ordered list in response to partial system failures and changes in the ordered list of strings received from the sorting logic. In the previous version of the mission controller, the string selection logic implicitly used an importance based ordering for string sorting logic. The string selection logic operates in batch mode in that it deploys/kills/redeploys groups of strings at the same time. Note that in this design, some deployed strings may be redeployed to use different resources.

When selecting the groups of strings to deploy/kill/redeploy, the string selection logic individually sends string deployment plan requests to the string binding logic. The string binding logic was previously called the low-level control. Upon receiving a string deployment request from the string selection logic, the string binding logic still generates a string deployment plan that is returned to the string selection logic, if such a deployment plan is possible. If a deployment plan is not feasible, then the binding logic returns a null plan.

In order to maintain the best possible estimate of the state of resources available to the mission controller, the string binding logic maintains an internal estimate of the current availability of resources. This estimate of resources availability is based on resource information input to the controller, and is updated as the binding logic generates deployment plans for the selection logic. Note that external information on the resources available to the mission controller is not updated instantaneously in real systems. (This is reflected in our updated Simulink model.) Therefore, the binding logic will generally have delayed information about true the availability of resources available to the mission controller. In order to further maintain the best possible estimate of the state of resource availability, the string binding logic is also notified by the string selection logic about strings that the selection logic plans to kill.

5.3.3.1 String Selection Logic

The string selection logic directs the (re)deployment and killing of the mission's strings in the order dictated by an ordered list of the mission's strings. This ordered list of strings is passed to the string selection algorithm as a parameter from the sorting logic. In order to operate, the string selection logic is assumed to have access to information about which of its strings are deployed, and the occurrence of pool failures. The string selection algorithm is presented in pseudo code in Figure 56 below.

During operation, the mission controller maintains four separate ordered lists of strings which have the same relative ordering as the mission-wide ordered string list.:

- Strings that are currently operating.
- Strings that are not deployed, but whose deployment may be feasible due information about the success of previous deployment operations.
- Strings to be killed during the next batch of string deployments.
- Strings to be (re)deploy during the next batch of string deployments with their deployment plans.

```
stringSelectionLogic
{       Given:
        L: the ordered list of strings.
        S: a list of deployed strings.

        From L and S, generate D, an ordered list of deployed, operating strings.
        From L and S, generate U, an ordered list of strings not deployed or operating.

        Define:
        K: a null list of deployed strings to kill.
        B: a null list of strings to (re)deploy with deployment plans.

        While U is non-empty
        {
                s = firstString(U);
                deploymentPlan = lowLevelLogic(s);
                if deploymentPlan != null
                {
                        B.add(s,deploymentPlan);
                        If K.contains(s)
                                K.remove(s);
                        U.remove(s);
                }
                else
                {
                        d = lowestValueString(D)
                        if s ahead of d in L
                        {
                                K.add(d);
                                If B.contains(d)
                                        B.remove(s);
                                D.remove(d);
                                U.addInOrder(d,L);
                        }
                        else
                        {
                                U.remove(s);
                        }
                }
        }
        Kill all strings in K.
        Bind all strings in B to their resources.
}
```

Figure 56: String selection Logic.

The string selection logic incrementally attempts to generate deployment plans for its internal list of undeployed strings by querying the string binding logic. As string deployment plans are successfully generated for strings on this undeployed list, those strings and their deployment plans are placed on the list of strings to be (re)deployed.

If a deployment plan cannot be generated for a string, the string selection algorithm may kill a string to free up resources, depending on the relative ordering of the strings on the list generated by the sorting logic. To do this, the selection logic tests to see if the last string in the sorted list of operating strings is behind the undeployable string in the composite sorted list of strings. If so, the last string in the sorted list of operating strings is added to the kill list and a notification of this kill is sent to the binding logic that the killed string's resources are free. The killed string is then also added to the list of strings to whose deployment may be feasible because it may be possible to redeploy the string later on different resources.

The logic continues to kill low ordered strings until enough resources are freed up to deploy the string. When the string binding logic successfully generates a deployment plan for the string, this string is added to the list of strings to deploy along with its plan and the string is removed from list of strings to deploy. If a deployment plan cannot be generated for a string and there are no strings which could be killed that would be free up resources for the undeployed string, the string is removed from the list of strings whose deployment may be feasible and not considered for later deployment.

The string selection logic continues operation until the list of strings whose deployment may be feasible is empty. When the feasible string list is empty, the string selection algorithm outputs the list of strings to be killed and the list of strings to be (re)deployed with their deployment plans. As strings are killed, the killed strings are removed from the list of operating strings and added to the list of strings whose deployment may be feasible in the order proscribed by the master string list.

5.3.3.2 Sorting Logic

The operations of the string selection logic (and hence the entire mission control behavior) is highly dependent on the sorted list of strings passed to the string selection logic. We compare and contrast three methods for sorting the strings:

1. Sorting based exclusively on the string's importance value. This method is called the *importance value method.*
2. Sorting based on the string's resource efficiency. This method is called the *resource efficiency method.*
3. Sorting based primarily on whether a string is in the set of most important strings and secondarily on a string's resource efficiency. This method is called the *two-order method.*

Algorithm A: Importance Value Sorting

The first string sorting method is also the simplest. Each string is assigned a importance value by the user. The strings are then sorted based on these importance values with the highest importance values placed first in the list. This was the method used in our initial refinement of the string selection logic.

Algorithm B: Resource Efficiency Sorting

The second sorting method is based on the intuition that rather than deploying the strings with the highest importance values, it may be desirable to deploy the strings that use resources most efficiently. We measure string resource efficiency as the ratio of a string's importance value to its resource usage as determined by the string resource usage measure. To demonstrate, suppose there are n strings with importance values $imp_1, imp_2, \ldots, imp_n$ and resource usage measures $RU_1, RU_2, \ldots, RU_n$. For these n strings, define their respective resource efficiencies as $\frac{imp_1}{RU_1}, \frac{imp_2}{RU_2}, \ldots, \frac{imp_n}{RU_n}$.

Algorithm C: Two-Order Sorting (aka "Sauerkraut")

For the two-order string sorting method, the strings are partitioned into sets of strings depending on their user assigned importance. Although it is possible to partition the strings into more than two sets, we partition the strings into the most important and non-most important sets. The strings in their respective subsets are then sorted based on their resource efficiencies and placed on the main list first in order of their importance set and secondly based on resource efficiency.

As an example of how the string orderings are used, consider a mission with 4 strings s_1, s_2, s_3, s_4 with importance values $imp_1 = 2, imp_2 = 3, imp_3 = 4, imp_4 = 4$ and resource usages of $RU_1 = 5, RU_2 = 6, RU_3 = 4, RU_4 = 10$. Therefore, $\frac{imp_1}{RU_1} = 0.4, \frac{imp_2}{RU_2} = 0.5, \frac{imp_3}{RU_3} = 1, \frac{imp_4}{RU_4} = 0.4$.

The importance value based ordering for this set of strings would be 4,3,2,1 (with 4 arbitrarily placed ahead of 3), the efficiency based string order would be 3,2,1,4 (with 1 arbitrarily placed ahead of 4), and the two-order string order would be 3,4,2,1 (where strings 3 and 4 are deemed to be the most important strings.)

5.3.3.3 Resource Usage Computation

As discussed above, we use a string resource usage measure to aid in the ordering of strings for the efficiency-based and two-order ordering methods. We have formulated an easily computable string resource usage heuristic measure similar to the resource utility measure used by the string binding logic in Section 5.3.2.3.

Suppose that a string has a applications and hence $a-1$ inter-application communication links. Suppose also that the applications require $c_1, c_2, \ldots, c_a$ computational resources locally and the links each require $b_1, b_2, \ldots, b_{a-1}$ megabits per second of bandwidth. We approximate the string's total computation resource usage to be $\left(\sum_{i=1}^{a}\sqrt{c_i}\right)^2$ and the string's communication resource usage to be $\left(\sum_{i=1}^{a-1}\sqrt{b_i}\right)^2$. The total resource usage is therefore the weighted sum:

$$RU = \left(\sum_{i=1}^{a-1}\sqrt{b_i}\right)^2 + w\left(\sum_{i=1}^{a-1}\sqrt{b_i}\right)^2.$$

This measure for resource usage attempts to quantify the relative usage of the generally incomparable computation resource usage $\left(\sum_{i=1}^{a}\sqrt{c_i}\right)^2$ and the communication resource usage $\left(\sum_{i=1}^{a-1}\sqrt{b_i}\right)^2$. Because these measures are generally incomparable, we use the relative tuning factor w to balance these values. The weighting factor w is a tunable parameter that can be adjusted depending on relative resource availability. For example, when computation resources are configured to be scarce, w should be relatively small, while when communication resources are configured to be scarce, w should be relatively large.

5.3.3.4 Simulation Model and Experiments

We further developed our large-scale, highly configurable Matlab/Simulink model of the ARMS system to objectively compare the relative benefits of using the three different methods of string ordering procedures in conjunction with the mission controller,. This simulation model is a significantly upgraded and expanded version of the simulation model we used during previous refinements.

For our simulation experiments, we configured the model to consist of a mission with 100 strings that can be deployed on pool and inter-pool link resources. When the mission controller performs a string deployment operation, there is a configurable actuation delay between the time the mission controller sends the actuation signal until the time the string is becomes operational. Lacking an exact real-world for this deployment delay, we approximated this value as 0.1sec. Similarly, the mission controller model has a configurable observation delay to better model the mission controller's observation of changes in the availability of system resources due to partial failures or other reasons. Lacking an exact real-world value for this observation delay, this value is approximated as 0.1sec. We also designed the model to account for the computation time the mission controller requires to generate string deployment plans. This delay is added to the deployment delay of strings in order to generate an *effective deployment delay*.

In the simulation model, the operating conditions of the strings are highly configurable. The computational and communication requirements of the 100 strings can be customized to model various mission scenarios as long as there are at least two applications in every string. The user-assigned importance values of the strings are also configurable and can be used as experimental parameters in simulation.

The amount of resources available to a mission can also be adjusted in the simulation model. In particular, the number of pools and how many applications can be run in each pool and be individually adjusted along with the amount of inter-pool bandwidth available to the mission's strings in the system's inter-pool communication links. It is not necessary that the pools and links have homogenous resource configurations. We simulate partial system failures in real-time in the model by removing all of a pool's nodes to model a complete pool failures, and bandwidth is removed from communication links to simulate inter-pool communication link failures.

Simulation Experiments

Using the large-scale Matlab/Simulink model of the ARMS system, we generated 129 experimental string deployment scenarios consisting of 100 strings, each with randomly chosen application lengths uniformly distributed between 2 and 11. Inter-application bandwidth requirements were randomly chosen to be either 1 or 2 megabits per second. The 100 strings were randomly assigned integer importance values with a uniform random distribution between 1 and 10, inclusive.

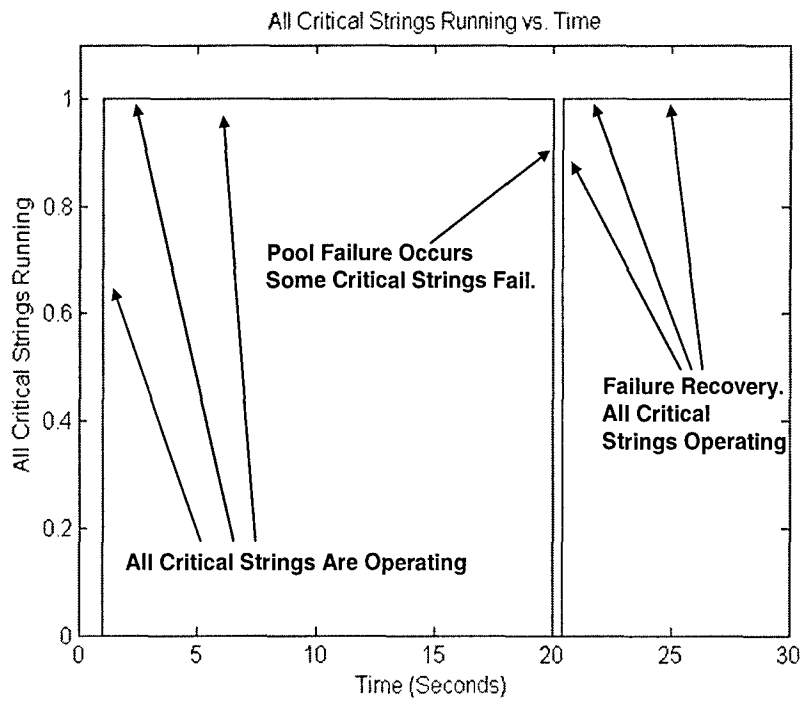

Figure 57: A detailed overview of Metric 1 warfighter value evolution during a simulation run.

For each scenario, the system had five operational pools at initialization with sufficient computational resources and bandwidth to deploy all strings. The pools were allocated computation resources such that after the failure of a specific pool, the mission controller would still have sufficient resources to deploy all strings, the failure of another individual pool would cause the mission to have only 80% of the resources required to deploy all strings, and the failure of a third individual pool would cause the system to have 50% of the resource to deploy all strings.

The Matlab/Simulink simulations were run such that the mission was given sufficient time to deploy all strings after initialization. After initialization was completed, a pool was failed, and the mission controller was allowed to complete its recovery operations in response to the pool failure. We recorded the amount of time for the mission controller to redeploy its most important strings (if any failed as a result of the pool failure), and we recorded the Metric 2 performance attained by the mission controller immediately before the failure and the Metric 2 performance after failure recovery operations completed.

The simulation was run 9 times for every scenario. During the first 3 simulation runs for every scenario, the pool that failed was chosen so that the mission would have sufficient resources after the failure to redeploy all strings. For these 3 simulations, each of the three string ordering methods was used once. During the second 3 simulation runs for every scenario, the pool that failed was chosen so that the mission would have 80% of the resources required to deploy all strings after the failure. For these 3 simulations, each of the three string ordering methods was used once. During the final 3 simulation runs for every scenario, the pool that failed was chosen so that the mission would have 50% of the resources required to deploy all strings after the failure. For these 3 simulations, each of the three string ordering methods was used once.

Figure 57 shows whether all of the most important strings in the system are running. In this figure, all most important strings are deployed soon after initialization, some most important strings fail when the pool fails at 20sec, and all failed most important strings fail soon after. Similarly, Figure 57 shows the sum of the importance values of the deployed strings. After initialization, the sum of importance values increases until it plateaus when all strings are deployed. After the pool failure, some strings fail and during recovery operations, some strings are killed to free up resources to redeploy other strings.

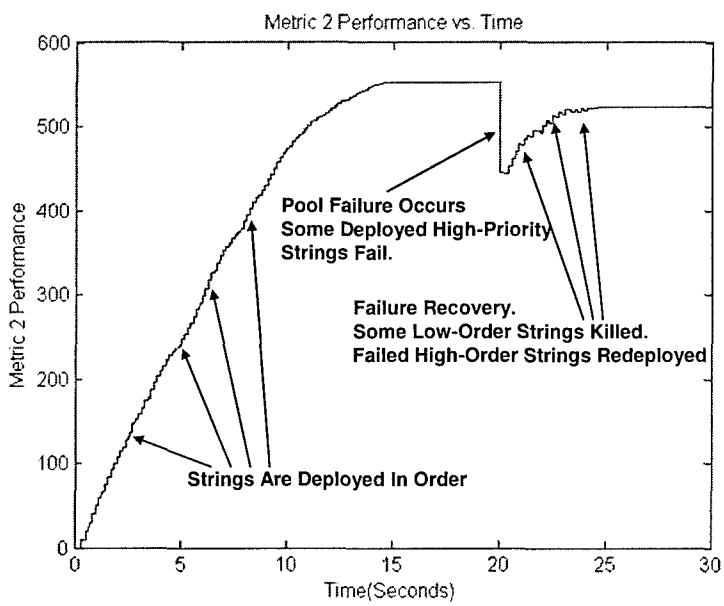

Figure 58: A detailed overview of Metric 2 performance evolution during a simulation run.

Experimental Results

From the simulation runs, we collected data on most important string recovery times for all simulations. Figure 59 contains a graph that demonstrates how the mean string recovery times for the simulation runs varies depending on the amount of resources available after the induced pool failure and the string ordering method used.

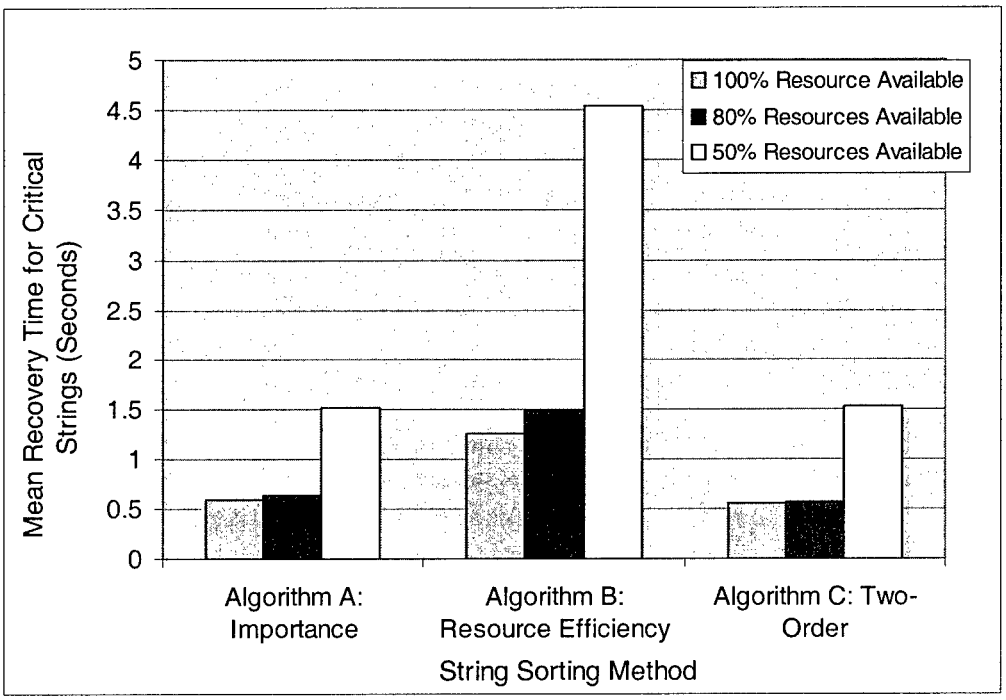

Figure 59: Mean String Recovery Time for Various String Ordering Methods for High and Low resource Availability.

As can be seen in Figure 59, the data shows that a controller using the two-order ordering method can achieve most important string recovery performance comparable to a controller using the importance value sorting method. Also note that the Metric 1 performance when the efficiency ranking was used is consistently less than the performance attained by the importance and two-order methods. This is due to the fact that the importance and two-order ranking methods give a higher priority to redeploying the system's most important strings while the resource efficiency method does not. Although the absolute measures of time used in this experiment are subjective, the data shows that a mission controller using the two-order ordering method can achieve most important string recovery performance comparable to a mission controller using the importance value sorting method.

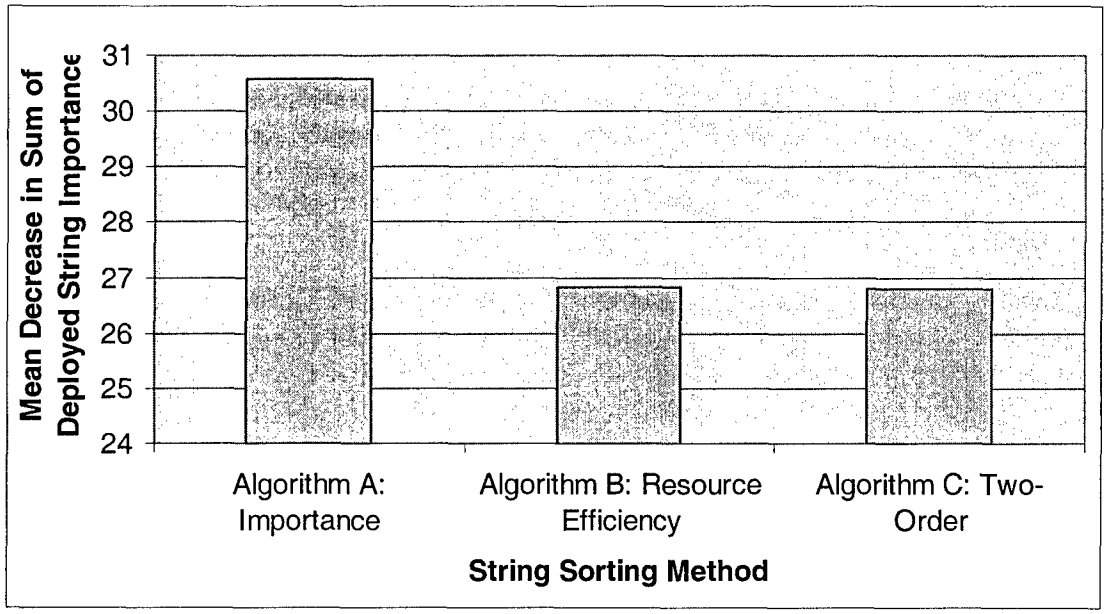

Figure 60: Decrease in Metric 2 Performance for Various String Ordering Methods During 80% Post-Failure Resource Availability.

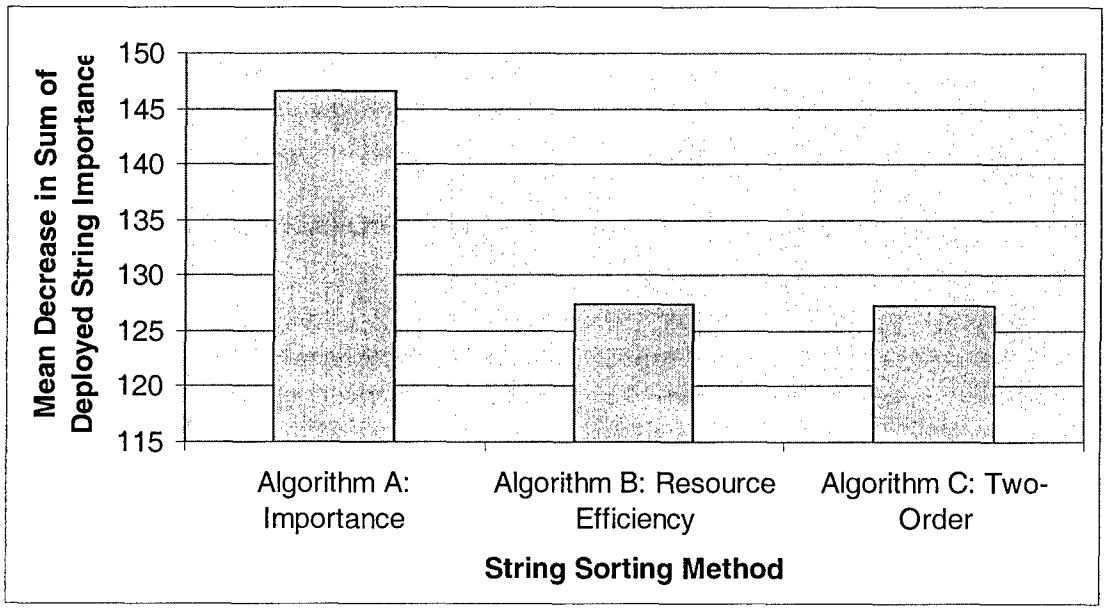

Figure 61: Decrease in Metric 2 Performance for Various String Ordering Methods During 50% Post-Failure Resource Availability.

From the simulation runs, we also collected data on the ability of the mission controllers to sufficiently regain lost Metric 2 performance after the failures. Figure 60 contains a graph that demonstrates how much lower the Metric 2 performance was than the pre-failure Metric 2 performance on average for the various ordering methods when there was 80% of the required resources available after failure. In these scenarios, the efficiency and two-order ordering methods performed comparably and better than the importance value based ordering method. The collected data also shows that the two-order and efficiency ordering method were able to deploy on the order of one more string (out of 100 strings) during failure recovery than the importance value based method.

Figure 61 contains a graph that demonstrates how much lower the Metric 2 performance was than the pre-failure Metric 2 performance on average for the various ordering methods when there was 50% of the required resources available. As can be seen from the graph, the efficiency and two-order ordering methods performed comparably and consistently better than the importance value based ordering method. The collected data shows that the two-order and efficiency ordering method were able to deploy on the order of four more strings (out of 100 total strings) during failure recovery than the importance value based method.

The Metric 2 simulation data clearly shows that the two-order ordering method can be used successfully to attain the desired behavior of both the importance value ordering under normal conditions and the efficiency order under constrained conditions. Additionally, there were no observed drawbacks associated with using the two-order ordering method. The two-order ordering method can be computed very efficiently and requires no special modifications of the previously established string selection logic.

5.3.4 Dynamic Programming Algorithm for Mission String Selection

Beyond our simple heuristic approaches to string ordering, we also attempted to use a dynamic programming approach to string selection. Among the three computationally efficient string ordering methods presented above, the resource efficiency algorithm achieved the highest Metric 2 performance during our Matlab/Simulink simulation experiments when used in conjunction with the string selection logic to deploy strings. However, it is still unknown how well the combined behavior of the string ordering and selection heuristics perform with respect to some optimal as measured by Metric 2. To formalize the optimal string selection problem at a high level of abstraction, the mission string selection problem is reducable to a multi-constrained knapsack problem [22]. The multi-constrained knapsack problems can be stated as selecting a subset of items with assigned values that satisfy a set of knapsack capacity constraints and maximize the total value of the selected items.

Formally, suppose we are given n items and m capacity constraints. The n items should be assigned binary values $x_1, x_2, \ldots, x_n$ to represent whether the n items are placed in the knapsack. The variables $v_1, v_2, \ldots, v_n$ are the respective values of placing the n items in the knapsack. The multi-constraint knapsack problem is to assign binary values to $x_1, x_2, \ldots, x_n$ such that $\sum_{j=1}^{n} v_j x_j$ is maximized subject to the m capacity constraints.

For the m capacity constraints, the variables $C_1, C_2, \ldots, C_m$ represent the size of the capacity constraints, and s_{ij} is the size of capacity i of item j. Therefore $s_{ij}x_j$ represents how much of capacity i is used by item j if it is selected to be placed in the knapsack. If $\sum_{j=1}^{n} s_{ij}x_j \leq C_i$, then the i^{th} capacities of all of the selected items fall within the constraint C_i.

Therefore, the formal statement of the multi-constrained knapsack problem is:

Maximize $\sum_{j=1}^{n} v_j x_j$

Subject to $\sum_{j=1}^{n} s_{ij}x_j \leq C_i \quad \forall i \in \{1,\ldots,m\}$ and $x_j \in \{0,1\} \quad \forall j \in \{1,\ldots n\}$

A special case of the *multi-constrained* knapsack problem is the simpler single-constraint knapsack problem [4] where there is only one constraint and hence $m = 1$. (The single-constraint knapsack problem is commonly referred to as the "knapsack problem")

For the single-constraint knapsack problem, suppose you are given a set of n items, with values $v_1, v_2, \ldots, v_n$ and sizes $s_1, s_2, \ldots, s_n$. The problem is to choose the subset of items that can be put into a knapsack with capacity C so that the total value of the items in the knapsack is maximized. Formally:

Maximize $\sum_{j=1}^{n} v_j x_j$

Subject to $\sum_{j=1}^{n} s_j x_j \leq C$ and $x_j \in \{0,1\} \quad \forall j \in \{1,\ldots n\}$

Methods developed to solve the single-constraint knapsack problem can be used to solve the mission-level string selection problem when the dominant limiting capacity is the amount of free computational resources. (That is, when the inter-pool bandwidth is considered sufficiently abundant.) This reduction is performed as follows:

- The number of strings maps to the number of items, n
- The string importance values map to items' value, v
- The string computational resource requirements map to items' size, s
- The total computational resources map to knapsack capacity, C

Straightforward use of a *dynamic programming algorithm* finds the optimal solution to the single-constraint knapsack problem and thus can be used to solve our string selection problem. Therefore, dynamic programming can be used to optimally solve the string selection problem in time $(O(nC))$[4].

There are several benefits to understanding this algorithm. For one, it allows us to analyze the string selection problem formally and provides an upper bound to the problem. Secondly, we can compare the performance of the computationally efficient resource efficiency algorithm with the optimal dynamic programming solution as a baseline. Thirdly, it allows us to evaluate the effectiveness, benefits and costs of the dynamic programming algorithm as a runtime alternative in solving the mission string selection problem.

We are currently comparing these two algorithms without the integration of low-level string binding logic. The total *Importance Value* (sum of importance values of strings selected to be deployed) is used to measure its performance. We hypothesize that good total Importance Value will yield good Metric 2 performance, but we need better end-to-end performance evaluation. This will be addressed later in the context of the overall mission and system resource management.

The dynamic programming algorithm and resource efficiency algorithm were implemented in C++ to compare their performance and execution time. We have not yet implemented the dynamic programming algorithm in the Matlab/Simulink model.

5.3.4.1 Experiment Setup

[4] THE ALGORITHM RUNS IN PSEUDO-POLYNOMIAL TIME AND NOT POLYNOMIAL TIME BECAUSE C IS NOT POLYNOMIAL IN THE LENGTH OF THE INPUT TO THE PROBLEM.

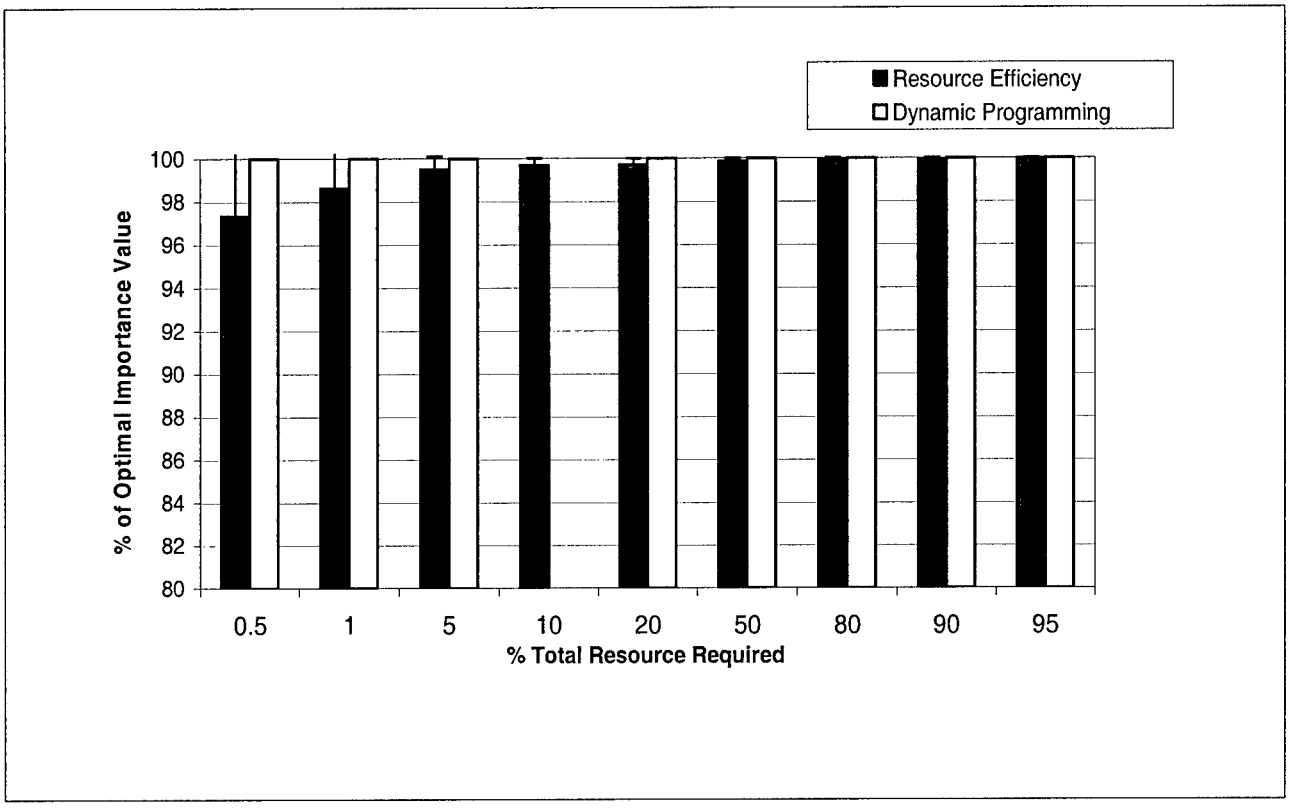

Figure 62 Comparison of the total Importance Value Achieved with Resource Efficiency and Dynamic Programming Algorithms under Different Resource Availability

We performed a large number of experiments to evaluate the performance of Dynamic Programming and Resource Efficiency algorithms under different resource availability, 0.5%, 1%, 5%, 10%, 20%, 50%, 80%, 90% and 95% of what is required to deploy all strings. At each resource level, 100 test runs were performed and average total importance value and execution time were measured and analyzed. In each test, there are 100 strings, each with its importance value and resource requirement picked randomly with a uniform distribution from 1 to 10 and 1 to 100 respectively.

5.3.4.2 Experiment Results

Because the dynamic programming algorithm produces optimal solutions, we use it as the baseline to evaluate the performance of the Resource Efficiency algorithm. Figure 62 shows that in our experiments the resource efficiency algorithm performed nearly as well as the dynamic programming algorithm at all resource levels, and especially at high resource availabilities (low resource contention). The percentage of importance value relative to the optimal only decreased slightly from 99.9% to 97.4% when the system resource went from 95% to 0.5% of what was required to deploy all strings. (Because it is optimal, the dynamic programming algorithm always achieves 100% of the importance value, but takes longer to do so due to the need for extra computational effort.)

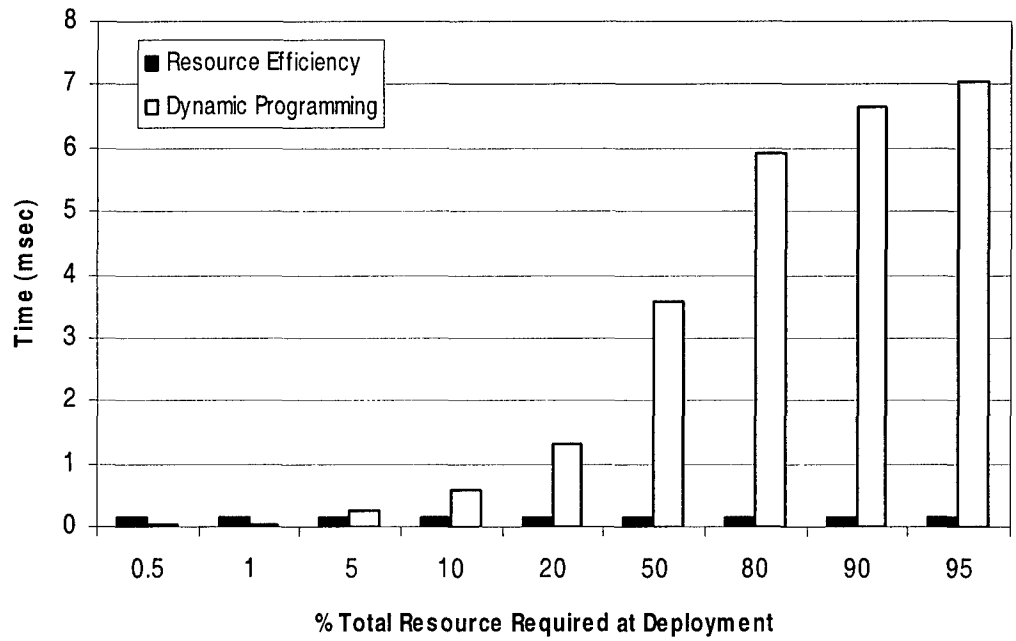

Figure 63 Comparison of the Execution Time of Resource Efficiency and Dynamic Programming Algorithms under Different Resource Availabilities

The resource efficiency algorithm can always be performed in time $O(n \lg n)$ as it is a sorting procedure and can be implemented very efficiently using known methods. As with our Matlab/Simulink experiments, we did not change the number of strings in these experiments. Therefore, because the execution time of resource efficiency algorithm is dependent only on the number of strings, in our experiments the average execution time of the resource efficiency algorithm is independent of the amount of free resources. This is reflected in Figure 63. Conversely, the execution time of the dynamic programming algorithm is highly dependent on the number of free resources. More precisely, the single-constraint knapsack problem is NP-hard and the dynamic programming algorithm runs in pseudo-polynomial time. In our experiments, the average execution time of the dynamic programming algorithm increased from around 24 microseconds when the system had 0.5% of its required resources to about 7 milliseconds at 95% of the required resources in our C++ implementation. This can be seen in Figure 63.

Even though the execution time increase is relatively large for the dynamic programming algorithm, the absolute time increase is relatively small (a few milliseconds) and might be acceptable in most cases. This is especially true when the number of strings or the resource space is relatively small. The additional computational cost of using a dynamic programming approach can be easily justified in order to achieve the optimal performance. The dynamic programming approach would be costly and impractical as an online procedure (in both time and memory) when the number of strings is very large (on the order of tens of thousands of strings) and/or the resource space is very large (sufficiently large to accept tens of thousands of strings), in which case the resource efficiency algorithm is a viable alternative to the optimal dynamic programming approach.

Although the resource efficiency algorithm usually performs well in practice, there are cases where the efficiency-based method performs noticeably worse than the optimal method. For a simple example, consider a system where the mission has access to one pool with 10 units of computational resources after a partial system failure causing some computational resources to fail. Further, assume that the mission has two strings, s_1 and s_2, with warfighter values $imp_1 = 2, imp_2 = 10$ and resource usage measures $RU_1 = 1, RU_2 = 10$. In this example, $\frac{imp_1}{RU_1} = 2, \frac{imp_2}{RU_2} = 1$, so string 1 would be deployed first using the resource efficiency ordering method. However, by deploying string 1, there would be insufficient resources to deploy string 2. As can be seen from inspection, the highest total importance value for this example would be attained if string 2 were deployed. Generally, situations where resource efficiency performs so poorly as compared to dynamic programming method generally occur when there are very limited resources. (This is evidenced by the general trend of resource efficiency performance as resource availability decreases as seen in Figure 62.) This therefore indicates that a general approach to attaining high Metric 2 performance without high computational cost would be to use a dynamic programming method when resources are scarce. During this situation, dynamic programming can be performed quickly. When resources are nearly sufficient, the resource efficiency method can be used because it is nearly optimal and dynamic programming cannot be performed quickly.

5.3.5 Multi-Mission Coordination

With our refined design of the mission controller, we turned our attention to the system controller, which we recast as a Multi-Mission-Coordinator (MMC). When selecting which strings to deploy, the amount of resources the mission is allowed to use is intended to be provided as a policy input from the MMC so that the MMC can direct the overall division of resources to the missions. Note that instead of being allocated specific resources, the refined mission controllers are given policy input as to how much resource they are allowed to use.

5.3.5.1 MMC Conops

When dividing the available system resources up amongst the missions, the MMC needs a way to predict what value the system would derive from allocating various amounts of resources to the missions. To do this, the MMC receives a lookup tables from each mission controllers that maps an approximation of the sums of the importance values of strings the mission controllers could deploy for their missions if given the ability to use various levels of resources. We further developed the mission controller design so that the lookup tables are generated and sent to the MMC by every mission controller at initialization and are based on user-commanded mission goals. The lookup tables are also intended to be updated regularly whenever a mission receives a command directive to refine its local behavior based on the relative importance values of the missions and its strings. Figure 64 contains a schematic of MMC operation which indicates that the lookup tables of missions' resource-value mappings.

When the lookup tables are sent to the mission controller, the resource levels listed in the lookup table are quantized based on the levels of quality of service provided by the missions for deploying groups of strings. Because the deployment of the missions' most important/critical strings is necessary for minimal mission operation, the lowest quantization level of the lookup tables correspond to the resources necessary to deploy just the most important/critical strings for each of the missions. The other quantization levels in the mission lookup tables are dependent upon the context of the mission and could be used to tune the operation of the MMC when dividing system resources among the missions.

When the MMC has the lookup tables from the mission controllers and given information about the availability of resources in the system, the MMC needs to decide how much resources should be provided to every mission controller in order to guarantee all critical strings can be run and to maximize the total value of all strings that can be deployed. The problem of the MMC allocating resources to the missions can be formalized as a multiple-choice knapsack problem. This problem is discussed in more detail below in the subsection entitled "Multi-Choice Knapsack and Dynamic Programming".

The MMC could use any number of algorithms to compute the most efficient division of resources among the active missions based on information from the lookup tables and resource efficiency. We found the dynamic programming algorithm to be very effective and efficient considering the relatively small numbers of missions that we are using.

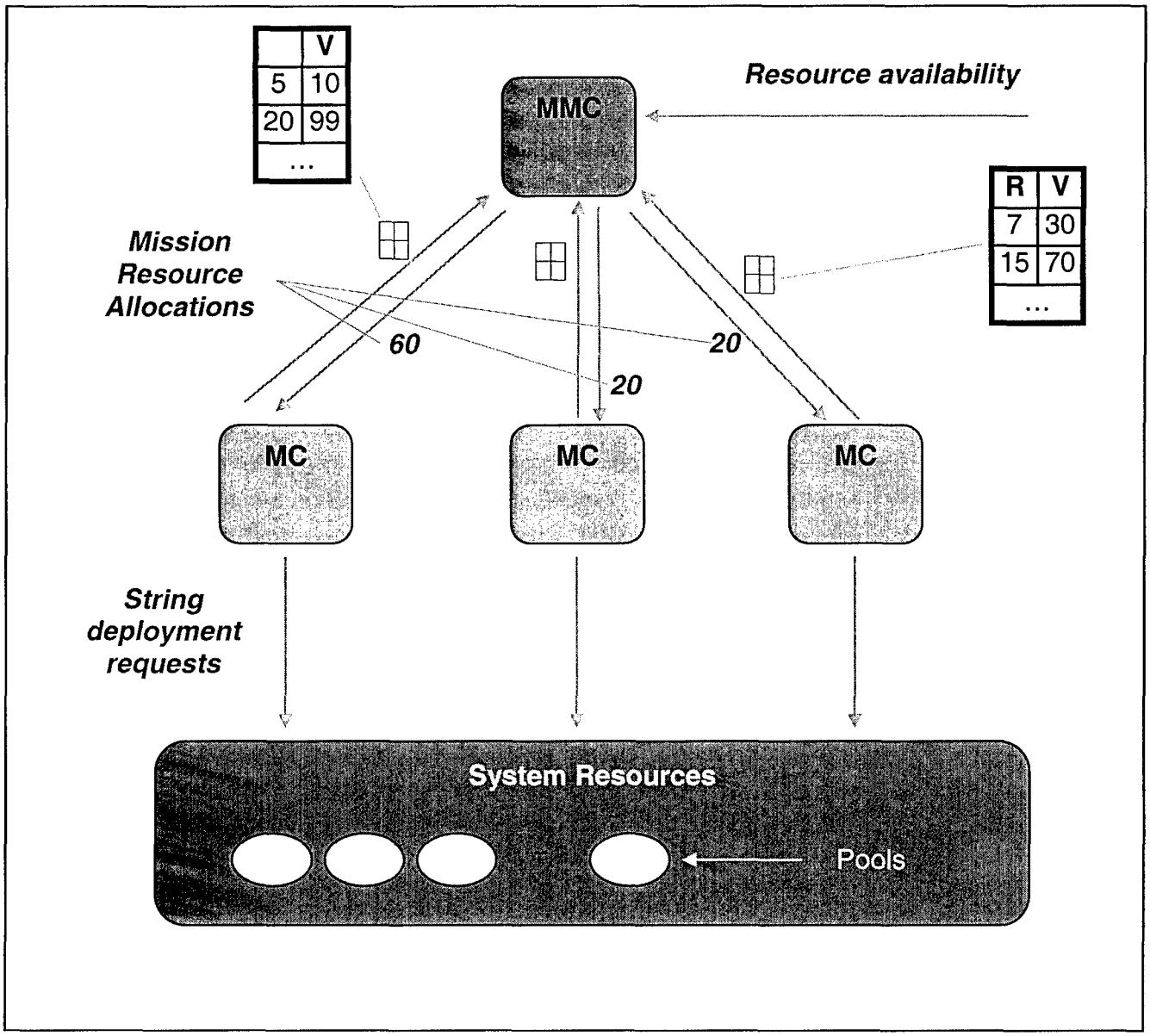

Figure 64: MMC Conops

Once the MMC computes the division of the resources amongst the missions, the MMC communicates to the mission controllers how much of the computation resources they are each allowed to use. This communication is indicated in the schematic of the MMC operation in Figure 64.

When the mission controllers have information about how much resources they are allowed to use, they make string deployment and killing requests to the system to deploy their strings that would deploy their critical strings and maximize the sum of the warfighter values of their deployed strings. The mission controllers receive no information about the allocation of resources to other missions. Therefore, on the occurrence of significant system events such as partial system failures or command directives, the mission controllers make all of their local resource allocation decisions under the assumption that their total resource allocation hasn't changed unless they receive updated information from the MMC.

5.3.5.2 Dynamic and Static Multi-Mission Coordinators

To explore the abilities of the MMC, we designed two MMC's to explore two variations of the MMC CONOPS. The first MMC, called the dynamic MMC, continually updates the allocation of resources to the mission controllers as system behavior evolves. Both MMC's used the dynamic programming algorithm to divide up system resources among the missions. After initialization, the dynamic MMC responds to system events such as partial system failures, command directives and changes in string values, among others by adjusting the missions' resource allocations. The static MMC does not adapt the resource allocations to the missions after initialization.

As previously stated, the static MMC is used as a proxy for a hypothetical baseline behavior that could be exhibited by the Program of Record (PoR). Although the static MMC does not adjust the high-level allocation of resources between the missions, the mission controllers continually attempt to dynamically use their full allocation of resources. We hypothesized that a system using this MMC could have very bad performance during failure recovery. Due to the operation of the mission controllers, a mission will not kill a string if it thinks it has sufficient resources to keep it deployed.

Unfortunately, because the static MMC does not adjust the allocation of resources to the missions, the mission may not know to kill its strings to free resources for other missions when the static MMC is used. Additionally, if a mission does kill its strings to free resources, other mission may take those resources before first mission could start its strings. These scenarios lead to situations where the missions attempt to use much more resources than it should and not all critical strings are recovered. We demonstrated this in our Matlab/Simulink simulations of the system using the static and dynamic MMC's

5.3.5.3 Multi-Choice Knapsack and Dynamic Programming

To formalize the optimal string selection problem at a high level of abstraction, the MMC resource division problem is reducible to a multi-choice knapsack problem[21]. The multi-choice knapsack problems can be stated as making a set of m choices where for each choice, an item has to be chosen, each with an assigned value and capacity requirements. The set of all chosen items over all choices has to satisfy a set of knapsack capacity constraints and maximize the total value of the selected items.

Formally, suppose n items are partitioned up into m classes, with N_k items in the class k for $k \in 1,\ldots,m$. The n items are assigned binary values $x_1, x_2, \ldots, x_n$ to represent whether the n items are chosen to be placed in the knapsack. The variables $v_1, v_2, \ldots, v_n$ are the respective values of placing the n items in the knapsack. The multi-choice knapsack problem is to assign binary values to $x_1, x_2, \ldots, x_n$ such that $\sum_{j=1}^{n} v_j x_j$ is maximized subject to the capacity constraint and exactly one item in a given class is chosen and assigned binary value 1.

For the capacity constraint, the variable C represent the size of the capacity constraint, and s_j is the size of item j. Therefore $s_j x_j$ represents how much of the capacity is used by item j if it is selected to be placed in the knapsack. If $\sum_{j=1}^{n} s_j x_j \leq C$, then the capacity of all of the selected items fall within the constraint.

Therefore, the formal statement of the multi-constrained knapsack problem is:

Maximize $\sum_{j=1}^{n} v_j x_j$

Subject to $\sum_{j=1}^{n} s_j x_j \leq C$, $x_j = 1$ for exactly one item in each class and $x_j = 0$ otherwise.

We are currently using the dynamic programming algorithm to solve the Multi-Choice Knapsack problem and perform the MMC-level resource allocation actions. We previously used the dynamic programming algorithm very effectively to solve resource allocation problems at the mission control level of the resource allocation hierarchy.

5.3.5.4 Simulation Model and Experiments

We developed a large-scale, highly configurable Matlab/Simulink model of the ARMS multi-mission system to objectively compare the behavior of the static and dynamic MMC's. This simulation model is a significantly upgraded and expanded version of the simulation model in experiments in Section 5.3.3.

For our simulation experiments, we configured the model to consist of three missions with 100 strings each that can be deployed on pool and inter-pool link resources. When the mission controllers perform string deployment operations, there is a configurable actuation delay between the time the mission controller sends the actuation signal until the time the string is becomes operational. Lacking an exact real-world for this deployment delay, we approximated this value as 0.1sec. Similarly, the mission controller models have a configurable observation delay to better model the mission controller's observation of changes in the availability of system resources due to partial failures or other reasons. Lacking an exact real-world value for this observation delay, this value is approximated as 0.1sec.

In the simulation model, the operating conditions of the strings are highly configurable. The computational and communication requirements of the strings can be customized to model various mission scenarios as long as there are at least two applications in every string. The user-assigned importance values of the strings are also configurable and can be used as experimental parameters in simulation.

The amount of resources available to the multi-mission system can also be adjusted in the simulation model. In particular, the number of pools and how many applications can be run in each pool and be individually adjusted along with the amount of inter-pool bandwidth available to the mission's strings in the system's inter-pool communication links. It is not necessary that the pools and links have homogenous resource configurations. We simulate partial system failures in real-time in the model by removing all of a pool's nodes to model a complete pool failures, and bandwidth is removed from communication links to simulate inter-pool communication link failures.

Simulation Experiments

Using the large-scale Matlab/Simulink model of the ARMS system, we generated 100 experimental string deployment scenarios consisting of 3 missions off 100 strings, each with randomly chosen application lengths uniformly distributed between 2 and 11. Inter-application bandwidth requirements were randomly chosen to be either 1 or 2 megabits per second. The 100 strings were randomly assigned integer importance values with a uniform random distribution between 1 and 10, inclusive. To generate the lookup tables generated by the mission controllers and sent to the MMC, we randomly grouped the missions' string sets into 10 quantization levels.

For each scenario, the system had five operational pools at initialization with sufficient computational resources and bandwidth to deploy all strings. The pools were allocated computation resources such that after the failure of a specific pool, the mission controller would cause the mission to have only 80% of the resources required to deploy all strings. The failure of two pools would cause the system to have 60% of the resource to deploy all strings, the failure of three pools would cause the system to have 40% of the resource to deploy all strings and the failure of four pools would cause the system to have 20% of the resource to deploy all strings.

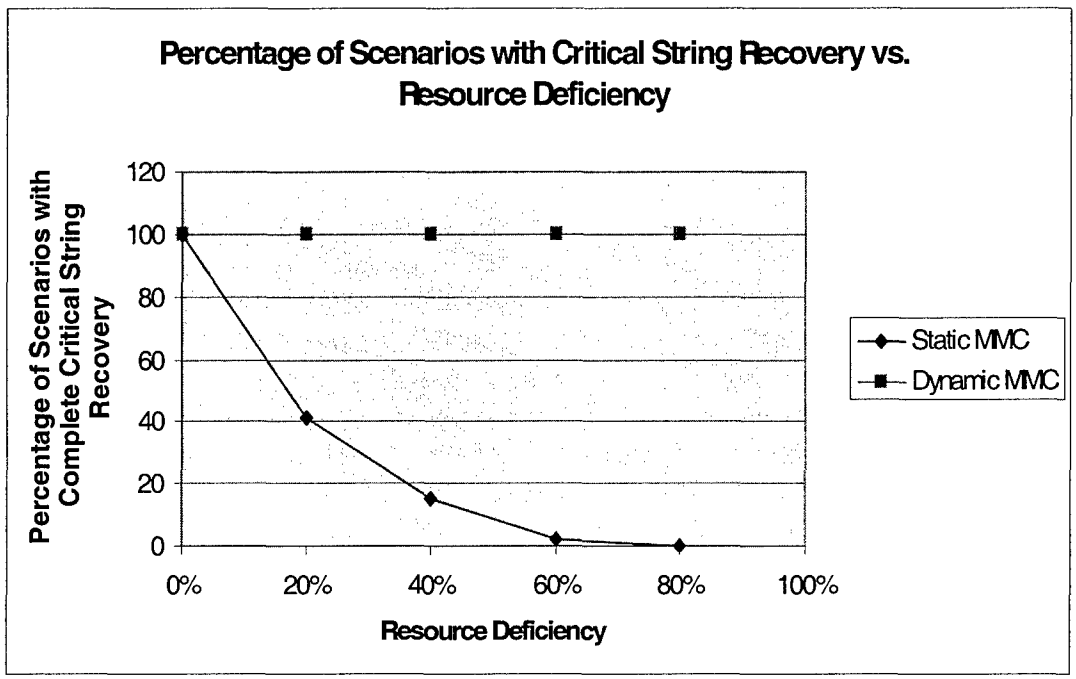

Figure 65: Percentage of Scenarios with Critical String Recovery vs. Resource Deficiency for Static and Dynamic MMC's.

The Matlab/Simulink simulations were run such that the MMC and mission controllers were given sufficient time to deploy all strings after initialization. After initialization was completed, a set of pools was failed, and the mission controller was allowed to complete its recovery operations in response to the pool failure. We recorded the amount of time for the mission controller to redeploy its most important strings (if all were able to recover as a result of the pool failure), and we recorded the Metric 2 performance attained by the mission controller immediately before the failure and the Metric 2 performance after failure recovery operations completed.

The simulation was run 8 times for every scenario. During the first 4 simulation runs the dynamic MMC was used and 1, 2, 3 or 4 of the pools were failed for each of the scenarios, respectively. During the second 4 simulation runs the static MMC was used and 1, 2, 3 or 4 of the pools were failed for each scenario, respectively.

Experimental Results

From the simulation runs, we collected data on whether the most important string were able to recover during the simulations. Figure 65 contains a graph that demonstrates how the percentage of critical string recovery varies depending on the amount of resources available after the induced pool failure and the MMC method used. As can be seen in Figure 65, the data shows that a controller using the dynamic MMC method can achieve acceptable most important string recovery performance. However, the static MMC was consistently unable to recover critical strings during low resource deficiency, and almost never able to recover critical strings during times of high resource deficiency. This is because the mission controllers in the system with a static MMC do not know to kill some strings to release resources for their more critical strings.

From the simulation runs, we also collected data on the ability of the mission controllers to sufficiently regain lost Metric 2 performance after the failures. Figure 66 contains a graph that demonstrates how the ratio of Metric 2 performance for systems using the static and dynamic MMC's vary with resource deficiency. As can be seen from the graph, as resource deficiency increases, the dynamic MMC is able to achieve over 2x performance gains over the static MMC.

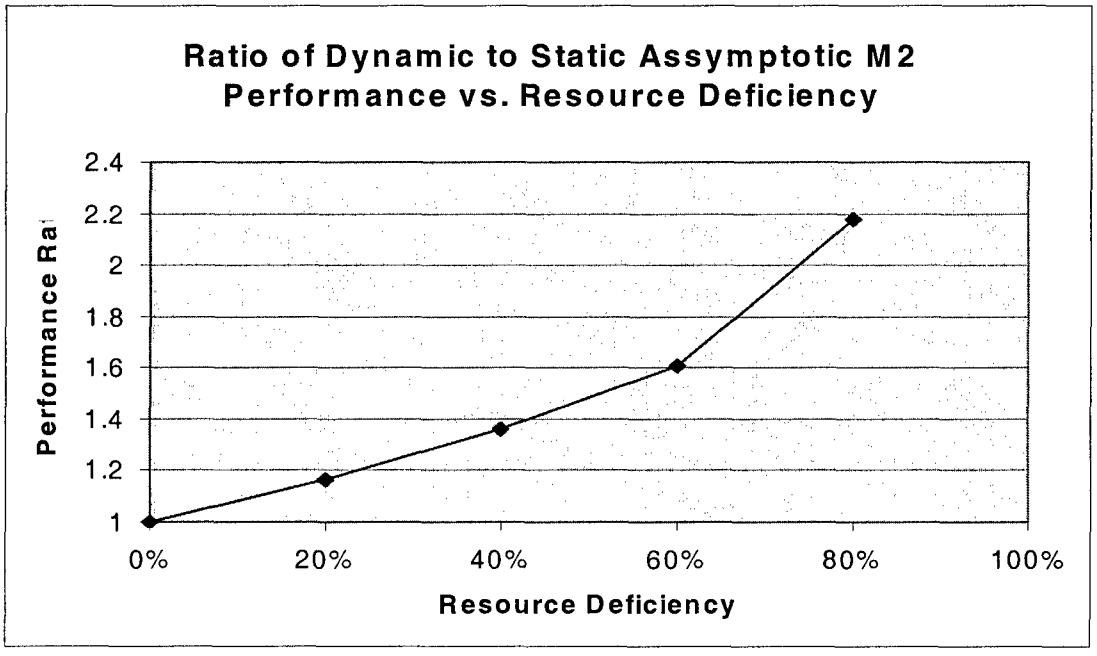

Figure 66: Ratio of Static to Dynamic MMC M2 Performance as Determined by Resource Deficiency

5.4 Transition of DRM Algorithms into ARMS Gate Test 4

As part of the ARMS Gate Test 4 efforts, we implemented our mission-level dynamic resource technology in the RACE environment. The team members from Raytheon and Lockheed Martin provided the RACE environment to test mission-level resource control strategies. Our goal was to demonstrate a two-order ***importance and resource efficiency planner*** that shows improvement over the baseline ***dynamic planner*** with respect to Metric 2, with no significant difference in Metric 1. Our threshold for improvement was 25% improvement, and our goal was 100% improvement.

Our team members had already implemented a baseline mission control system, and our importance-based mission control algorithm. Our two-order mission control technology was used as part of the icing efforts. In particular, we implemented our two-order DRM algorithm and ran tests of the system using this algorithm. We also designed the test scenarios that these experiments would be run on. Gate Test 4 letter-of-the-law (LoL) experiment's baseline ***dynamic planner*** was tuned to maintain operation of all high importance application strings and run as many of the successively next highest importance application string after node failures. This planning technique maximizes metric 1 but doesn't necessarily maximize metric 2.

5.4.1 Simulation Studies

In order to better suggest test scenarios for our team members that would demonstrate the benefits of our DRM algorithms. We used our Matlab/Simulink simulation environment to test our algorithms in proposed scenarios. Samples of this simulation data can be seen in Figure 67 and Figure 68.

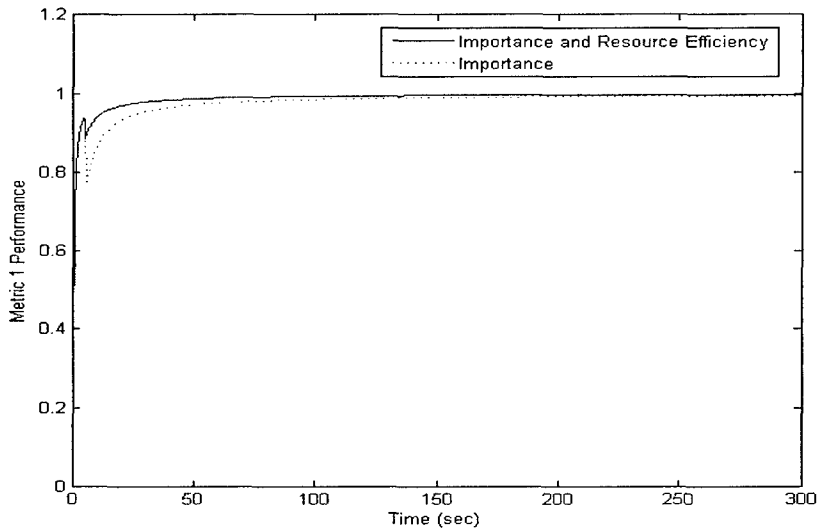

Figure 67: Metric 1 Performance of the Two-Order and Importance Algorithm in Simulink Simulation.

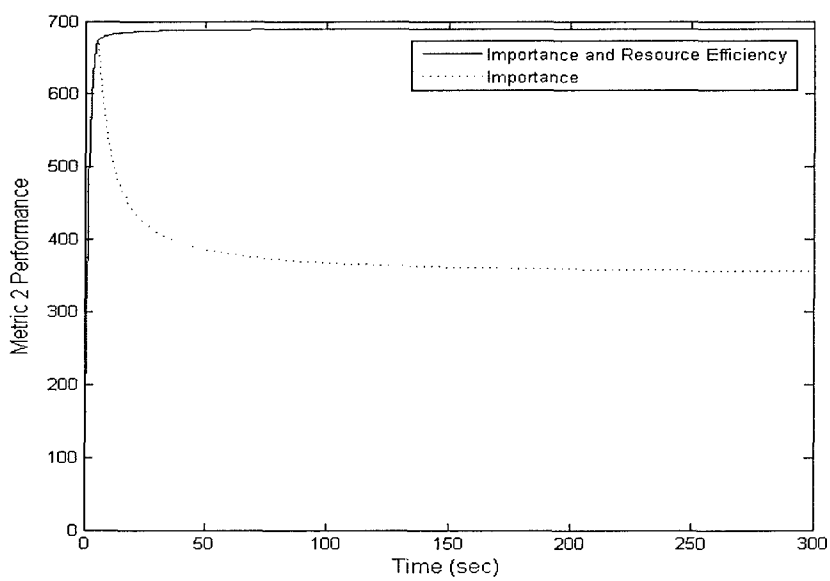

Figure 68: Metric 2 Performance of the Two-Order and Importance Algorithm in Simulink Simulation.

5.4.2 Results:

With the help of our team members, we ran the icing experiments in the RACE environment. Samples of the data output in these experiments can be seen in Figure 69 and Figure 71.

The compiled results of these experiments can be seen in Figure 70 and Figure 72. The icing experiments show that we could achieve greater survivability of war-fighting functionality (Metric 2) using resource allocation based upon the two-order mission control algorithm. In our test scenario, we measured equivalent values with metric 1 but twice the value (~100% improvement) with metric 2. We also observed that there were other scenarios (including the scenarios used for the LoL tests) in which the two algorithms performed equally well.

Our results in this experimentation were consistent with results obtained through Matlab Simulink simulation before the test runs. This not only demonstrates the usefulness of the DRM algorithms, but also the viability of our Matlab/Simulink simulation tool to quickly obtain simulated resource management data.

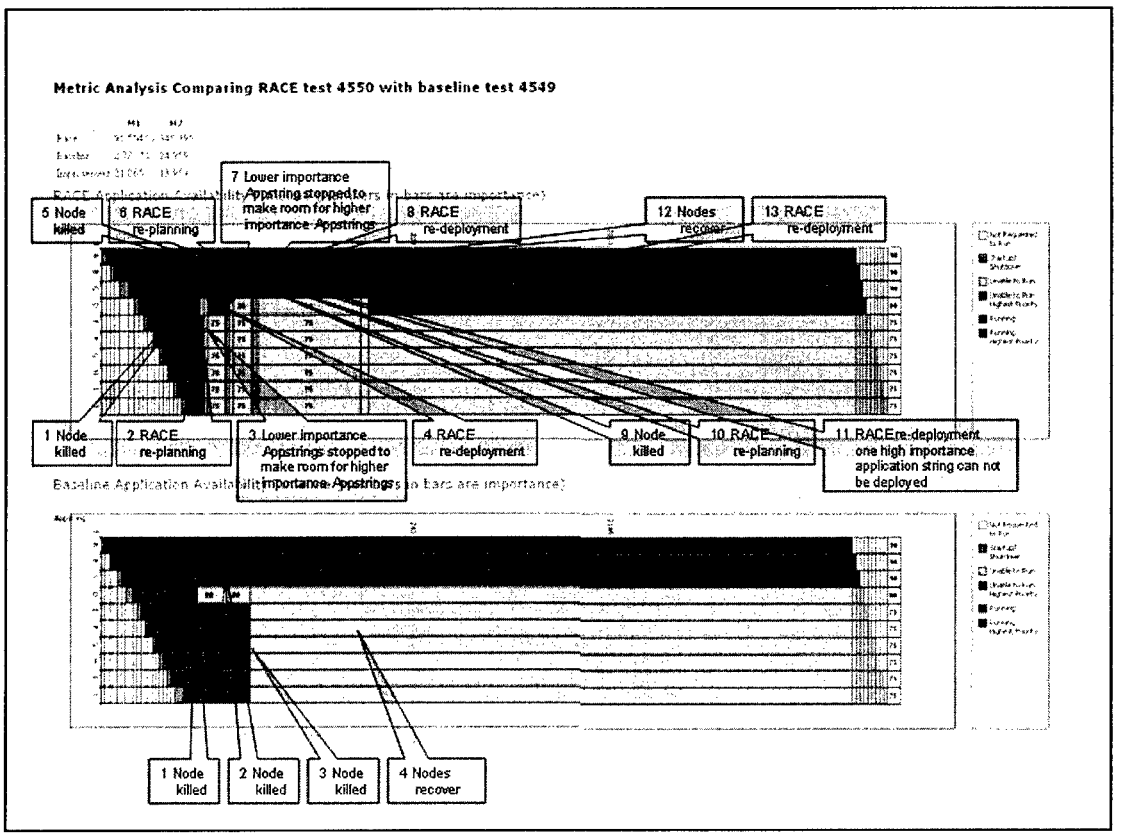

Figure 69 Sample Test Run Analysis in Race using Importance-based Mission Control

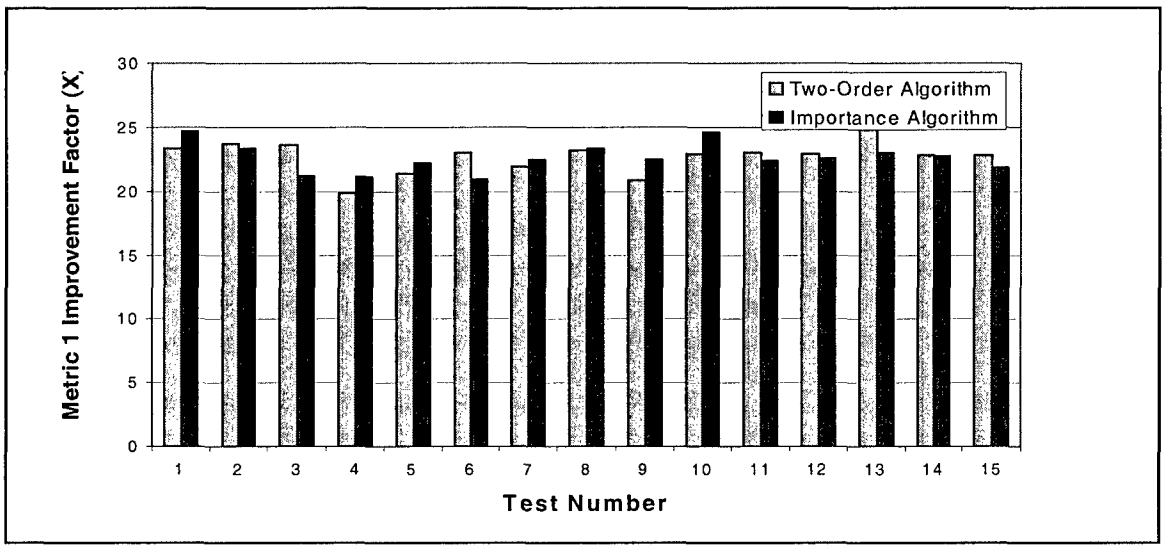

Figure 70: Metric 1 Performance Improvement over POR Baseline

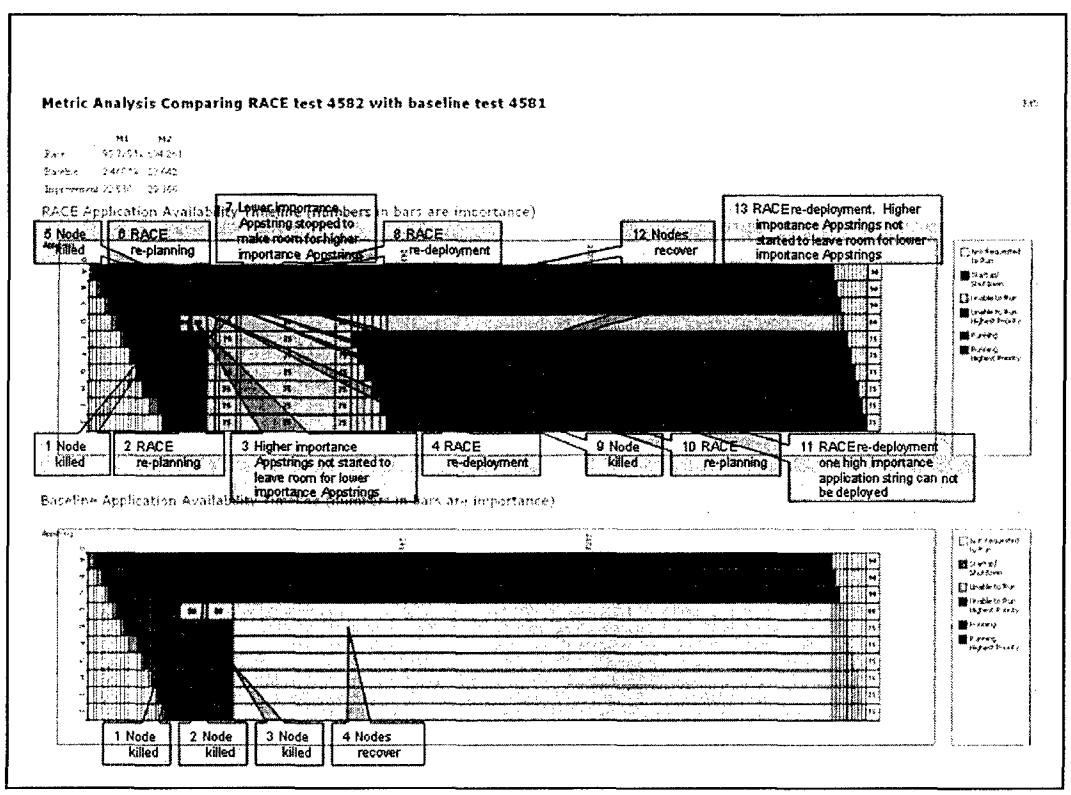

Figure 71 Sample Test Run Analysis in Race using Two-Order Mission Control

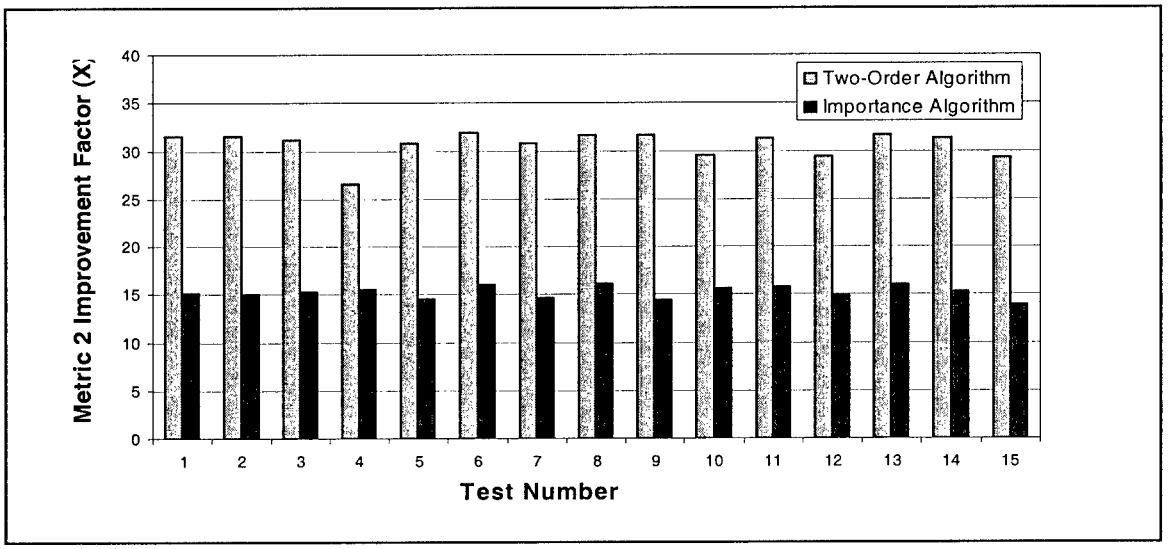

Figure 72: Metric 2 Performance Improvement over POR Baseline

6. Additional Programmatic Information

6.1 Chronological List of Publications under the ARMS Project

Publication 1	
Title	Detection and Reaction to Unplanned Operational Events in Large Scale Distributed Real-Time Embedded Systems
Author(s)	Jianming Ye, Joe Loyall, Rick Schantz, Gary Duzan
Publication Date	4/4/2005
Publication Venue	International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2005)
Keywords	Dynamic adaptation, unplanned response
URL	
Comments	

Publication 2	
Title	Detection and Reaction to Unplanned Operational Events in Large Scale Distributed Real-Time Embedded Systems
Author(s)	Jianming Ye, Joe Loyall, Rick Schantz, Gary Duzan
Publication Date	4/4/2005
Publication Venue	Proceedings of the Workshop on Parallel and Distributed Real-Time Systems

Keywords	**Adaptive quality of service; detection; decision; reaction**
URL	
Comments	**Includes viewgraphs and presentation**

Publication 3	
Title	**Using Composition of QoS Components to Provide Dynamic, End-to-End QoS in Distributed Embedded Applications A Middleware Approach**
Author(s)	Praveen Sharma, Joseph Loyall, Richard Schantz, Jianming Ye, Prakash Manghwani, Matthew Gillen, and George T. Heineman
Publication Date	**12/31/2010**
Publication Venue	**IEEE Internet Computing Journal**
Keywords	**dynamic resource management**
URL	
Comments	

Publication 4	
Title	**A Hierarchical Control System for Dynamic Resource Management**
Author(s)	**K. Rohloff, J. Ye, J. Loyall, R. Schantz**

Publication Date	**4/4/2006**
Publication Venue	**12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)**
Keywords	**dynamic resource management, feedback control software**
URL	**http://www.rtas.org/.**
Comments	

•

Publication 5	
Title	**Toward an Approach for Specification of QoS and Resource Information for Dynamic Resource Management**
Author(s)	**Roy Campbell, Rose Daley, B. Dasarathy, Patrick Lardieri, Brad Orner, Rick Schantz, Randy Coleburn, Lonnie Welch, Paul Work**
Publication Date	**5/25/2004**
Publication Venue	**2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04), Toronto, Canada**
Keywords	**multi-layer resource management**
URL	
Comments	

Publication 6	
Title	Utility Functions and Control for Multi-Layer Resource Management
Author(s)	J. Ye, K. Rohloff, R. Schantz, J. Loyall
Publication Date	12/2/2005
Publication Venue	ARMS Wiki
Keywords	utility functions, multi-layer resource management, feedback control
URL	https://repo.isis.vanderbilt.edu/twiki/bin/view/AR
Comments	Initially published on the internal ARMS Wiki for program wide distribution

Publication 7	
Title	Quality Measures for Embedded Systems and Their Application to Control and Certification
Author(s)	Kurt Rohloff, Joseph Loyall, Richard Schantz.
Publication Date	4/4/2006
Publication Venue	Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), Workshop on Innovative Techniques for Certification of Embedded Systems
Keywords	software certification, adaptive reconfiguration
URL	

Comments	

Publication 8	
Title	A Hierarchical Control System for Dynamic Resource Management,
Author(s)	Kurt Rohloff, Jianming Ye, Joseph Loyall, Richard Schantz
Publication Date	4/7/2006
Publication Venue	2006 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)
Keywords	multi-layer dynamic resource management
URL	
Comments	

Publication 9	
Title	Adding Fault-Tolerance to a Hierarchical DRE System
Author(s)	Paul Rubel, Joseph Loyall, Richard Schantz, Matthew Gillen
Publication Date	6/14/2006
Publication Venue	Proceedings of Distributed Applications and Interoperable Systems: 6th IFIP WG 6.1 International Conference, DAIS

	2006
Keywords	fault tolerance mechanisms case study
URL	
Comments	

Publication 10	
Title	Providing Fault-Tolerant Management in a CCM DRE Environment
Author(s)	Paul Rubel, Joseph Loyall, Matthew Gillen, Aniruddha Gokhale, Jaiganesh Balasubramanian, Priya Narasimhan, Aaron Paulos
Publication Date	7/10/2006
Publication Venue	OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems
Keywords	fault tolerance, CCM, DRE
URL	
Comments	Presentation

Publication 11	
Title	Fault Tolerance in a Multi-Layered DRE System: A Case Study
Author(s)	Paul Rubel, Joseph Loyall, Richard Schantz, Matthew Gillen

Publication Date	**9/1/2006**
Publication Venue	**Journal of Computers, Vol. 1, Issue 6**
Keywords	**fault tolerance, multi-layer dynamic resource management, component middleware, DRE systems**
URL	**http://www.academypublisher.com/jcp/vol01/no06/ind**
Comments	

6.2 Chronological Review of ARMS Activities

Meeting or Presentation 1	
Meeting Name	**ARMS Kickoff**
Purpose	**To kickoff the ARMS program**
Start Date	**10/23/2003**
End Date	**10/23/2003**
Location	**Washington, DC**
Attendees	**Rick Schantz, Joe Loyall**
Titles of presentations that were made	**Adaptive Multilevel Middleware Systems (AMMS)**
Meeting or Presentation 2	
Meeting Name	**ARMS OEP Workshop**
Purpose	**Plan for the ARMS OEP and TD interaction**
Start Date	**11/13/2003**
End Date	**11/14/2003**
Location	**Arlington, VA**

Attendees	**Rick Schantz, Joe Loyall, Rich Shapiro**
Titles of presentations that were made	**Adaptive Multilevel Middleware Systems - Extending, Elaborating, and Building the Multi-Layer Architecture Concept of Operation**
Meeting or Presentation 3	
Meeting Name	**Visit with POR software architects**
Purpose	**To discuss the nature of the POR software architecture and resource management needs**
Start Date	**12/16/2003**
End Date	**12/16/2003**
Location	**Portsmouth, RI**
Attendees	**Joe Loyall**
Titles of presentations that were made	**None**
Meeting or Presentation 4	
Meeting Name	**DARPA ARMS Integration Milestone "Hotwash" meeting**
Purpose	**Raytheon led meeting discussing integration and transition activities**
Start Date	**4/29/2004**
End Date	**4/29/2004**

Location	**Arlington, Va**
Attendees	**Rick Schantz, Joe Loyall, Gary Duzan, Rich Shapiro, Balachandran Natarajan, Jeff Parsons**
Titles of presentations that were made	
Meeting or Presentation 5	
Meeting Name	**ARMS PI meeting**
Purpose	**Present progress and plans for the ARMS program**
Start Date	**4/30/2004**
End Date	**4/30/2004**
Location	**Arlington Va**
Attendees	**Rick Schantz, Joe Loyall, Gary Duzan, Rich Shapiro, Bala Natarajan, Jeff Parsons**
Titles of presentations that were made	**Elaborating and Building the Multi-Layer Dynamic Architecture Concept and Components**
Meeting or Presentation 6	
Meeting Name	**Design Workshop**
Purpose	**To flesh out the multi-pool and multi-resource manager design concepts**
Start Date	**6/3/2004**

End Date	**6/4/2004**
Location	**BBN, Cambridge, MA**
Attendees	**Schantz, Duzan, Shapiro, Lardieri, Mulholland, Daley, Work, Dasarathy, Coleburn, ...**
Titles of presentations that were made	**Various**
Meeting or Presentation 7	
Meeting Name	**ARMS PI Meeting**
Purpose	**To bring together all of the ARMS program participants for progress review and planning.**
Start Date	**7/22/2004**
End Date	**7/22/2004**
Location	**Arlington Va.**
Attendees	**Participants from all ARMS contractors and government users and sponsors**
Titles of presentations that were made	**BBN ARMS Components Status and Plans**
Meeting or Presentation 8	
Meeting Name	**OMG Workshop on Realtime and Embedded Systems**
Purpose	**To attend workshop and present interim project results**

Start Date	**7/19/2004**
End Date	**7/19/2004**
Location	**Washington DC**
Attendees	**B. Natarajan**
Titles of presentations that were made	**Using CCM to Develop Resource Status Service**
Meeting or Presentation 9	
Meeting Name	**ARMS Gate Metric Planning Workshop**
Purpose	**Planning Gate Metric experimentation and technical exchange with PoR program participants**
Start Date	**10/7/2004**
End Date	**10/8/2004**
Location	**Portsmouth, RI**
Attendees	**Richard Schantz, Gary Duzan**
Titles of presentations that were made	**Planning for the series of Gate Metrics experiments.**
Meeting or Presentation 10	
Meeting Name	**Gate Metric 2 Test and Evaluation in the OEP Laboratory**

Purpose	**Integration of ARMS software for establishing compliance with GM2**
Start Date	**12/18/2004**
End Date	**12/21/2004**
Location	**Portsmouth, RI**
Attendees	**Gary Duzan**
Titles of presentations that were made	
Meeting or Presentation 11	
Meeting Name	**Weekly Coordination Teleconferences**
Purpose	**Planning and status coordination**
Start Date	**10/5/2004**
End Date	**12/23/2004**
Location	**By Telephone**
Attendees	**Richard Schantz, Gary Duzan, Richard Shapiro, Joe Loyall**
Titles of presentations that were made	
Meeting or Presentation 12	

Meeting Name	**ARMS PI Meeting**
Purpose	**To discuss and plan ARMS technical activities**
Start Date	**1/13/2005**
End Date	**1/13/2005**
Location	**Arlington, Va.**
Attendees	**Rick Schantz, Gary Duzan**
Titles of presentations that were made	**ARMS Phase 1 Summary and Candidate ARMS Phase 2 Activities**
Meeting or Presentation 13	
Meeting Name	**Software Integration**
Purpose	**Test and evaluate ARMS software from multiple contractions**
Start Date	**1/5/2005**
End Date	**1/7/2005**
Location	**Portsmouth RI**
Attendees	**Gary Duzan et al**
Titles of presentations that were made	

Meeting or Presentation 14	
Meeting Name	**Software Integration**
Purpose	**Integrate, test and evaluate ARMS software for GM3**
Start Date	**1/25/2005**
End Date	**1/27/2005**
Location	**Portsmouth RI**
Attendees	**Gary Duzan et al**
Titles of presentations that were made	

Meeting or Presentation 15	
Meeting Name	**ARMS Phase 2 Kickoff**
Purpose	**To mark the beginning of ARMS phase 2**
Start Date	**5/3/2005**
End Date	**5/4/2005**
Location	**Arlington, Va.**
Attendees	**R. Schantz, J. Loyall, G. Duzan, A. Gokhale, N. Shankaran and J. Balasubramanian**
Titles of presentations that	**BBN Lessons Learned from Phase 1 and Plans for Phase 2**

were made	
Meeting or Presentation 16	
Meeting Name	**Common planning with Lockheed Martin**
Purpose	**Project planning and integration**
Start Date	**5/27/2005**
End Date	**5/27/2005**
Location	**Cherry Hill, New Jersey (Lockheed Martin ATL)**
Attendees	**R. Schantz, J. Loyall, P. Lardieri, Ed Mulholland**
Titles of presentations that were made	
Meeting or Presentation 17	
Meeting Name	**ARMS PI Meeting**
Purpose	**Progress Review and Working Groups**
Start Date	**9/27/2005**
End Date	**9/28/2005**
Location	**Arlington Va**
Attendees	**R. Schantz, J. Loyall, M. Gillen, P. Rubel, J. Ye, K. Rohloff**

Titles of presentations that were made	**Project Summary, Results and Plans Fault Tolerance Workshop Overview Supporting ACtive Replication in GT3 Utility Functions: Concepts and Design**
Meeting or Presentation 18	
Meeting Name	**ARMS Transition**
Purpose	**Planning for technology transition**
Start Date	**8/30/2005**
End Date	**8/30/2005**
Location	**Arlington Va**
Attendees	**J. Loyall, P. Work, P. Lardieri, J. Cross, et al**
Titles of presentations that were made	**ARMS Transition Opportunities**
Meeting or Presentation 19	
Meeting Name	**Project Coordination Meeting**
Purpose	**Planning and working development meeting with Vanderbilt and CMU**
Start Date	**7/20/2005**
End Date	**7/21/2005**
Location	**BBN Cambridge**

Attendees	**Doug Schmidt, Andy Gohkale, et al, BBN ARMS team**
Titles of presentations that were made	**various**
Meeting or Presentation 20	
Meeting Name	**Fault Tolerance Planning and Design**
Purpose	**Working meeting on GT3**
Start Date	**8/10/2005**
End Date	**8/11/2005**
Location	**BBN, Cambridge**
Attendees	**Aaron Paulos, Priya Nahrsimhan, Joe Slember, BBN ARMS team**
Titles of presentations that were made	**various**
Meeting or Presentation 21	
Meeting Name	**RT Java**
Purpose	**To discuss and review plans for transitioning RT Java**
Start Date	**7/19/2005**
End Date	**7/19/2005**

Location	**Raytheon, Sudbury Ma**
Attendees	**R. Schantz, P. Work, J. Cross, D. Schmidt, G. Thacker, et al**
Titles of presentations that were made	
Meeting or Presentation 22	
Meeting Name	**Gate Test 3 Planning**
Purpose	**To work out design, implementation and schedule details for 3 month gate test plan**
Start Date	**10/31/2005**
End Date	**11/1/2005**
Location	**BBN Technologies, Cambridge Ma.**
Attendees	**Loyall, Rubel, Gillen, Ye, Schantz, Poulos, Dasarathy ,** Balasubramanian, **...**
Titles of presentations that were made	**Numerous by BBN, Telcordia, Vanderbilt, CMU**
Meeting or Presentation 23	
Meeting Name	**STAG Telcons**
Purpose	**Bi-weekly ARMS management team planning sessions**
Start Date	**10/11/2005**

End Date	**12/13/2005**
Location	**electronic meeting**
Attendees	**Schantz, Loyall, Cross, Howard, Lardieri, Thacker, Schmidt, Work, Legault**
Titles of presentations that were made	
Meeting or Presentation 24	
Meeting Name	**ARMS-Navy Tech Transfer Workshop**
Purpose	**To exchange and coordinate areas of concentration and plans for identified areas of technology transition.**
Start Date	**2/28/2006**
End Date	**2/28/2006**
Location	**Raytheon DDX Integration Center, Navy Yard Washingto DC**
Attendees	**Rick Schantz, Joe Loyall, et al**
Titles of presentations that were made	
Meeting or Presentation 25	
Meeting Name	**Raytheon/BBN Tech exchange workshop**
Purpose	**To present ARMS technical approaches to Program of Record**

	staff in the area of fault tolerance
Start Date	**1/18/2006**
End Date	**1/18/2006**
Location	**Portsmouth RI**
Attendees	**Rick Schantz, Joe Loyall, Paul Rubel, Matt Gillen, Paul Work, Rich Lescroat, Al Peckham**
Titles of presentations that were made	**Approach to Fault tolerance for MLRM (GT3); A design and analysis for node failure detection;**
Meeting or Presentation 26	
Meeting Name	**Bi-weekly ARMS management telcons**
Purpose	**To coordinate and assess progress on ARMS program activities**
Start Date	**1/1/2006**
End Date	**3/31/2006**
Location	**various**
Attendees	**Schantz, Loyall, et al**
Titles of presentations that were made	
Meeting or Presentation 27	
Meeting Name	**ARMS Principal Investigators Meeting**

Purpose	**To report progress and plan next stages of program wide objectives**
Start Date	**4/11/2006**
End Date	**4/13/2006**
Location	**Arlington Va.**
Attendees	**Schantz, Loyall, Gillen, Rohloff, Ye, Manghwani, et al**
Titles of presentations that were made	**DARPA:ARMS Phase II PI Meeting BBN Technologies Team Report**
Meeting or Presentation 28	
Meeting Name	**Raytheon Site Visit**
Purpose	**To discuss details of potential technology transfer**
Start Date	**5/11/2006**
End Date	**5/11/2006**
Location	**Portsmouth, RI (Raytheon)**
Attendees	**R. Schantz, M. Gillen, R. Lescroart, A. Peckham, P. Work**
Titles of presentations that were made	**BBN's Node Failure Detection Capability**

Meeting or Presentation 29	
Meeting Name	**Transition Review**
Purpose	**To discuss arms transition activities**
Start Date	**6/26/2006**
End Date	**6/26/2006**
Location	**BBN, Cambridge Ma.**
Attendees	**R. Schantz, M. Gillen, Capt. Chris Earl, SRI**
Titles of presentations that were made	**Informal discussions**
Meeting or Presentation 30	
Meeting Name	**Management Telcons**
Purpose	**Bi Weekly progress and planning telcons among senior ARMS participants**
Start Date	**4/1/2006**
End Date	**6/30/2006**
Location	**telcon, bi weekly**
Attendees	**Key PIs , DARPA PM, advisors**
Titles of presentations that	

were made	
Meeting or Presentation 31	
Meeting Name	**Management Telcons**
Purpose	**Bi-weekly (or more frequent) and planning telcons among senior ARMS participants.**
Start Date	**7/1/2006**
End Date	**9/26/2006**
Location	**Telcon, bi-weekly**
Attendees	**Key PIs, DARPA PM, advisors**
Titles of presentations that were made	
Meeting or Presentation 32	
Meeting Name	**Integration meeting for Gate Test 4**
Purpose	**Working on utility function and controller algorithm enhancements for gate test 4**
Start Date	**8/24/2006**
End Date	**8/24/2006**
Location	**Raytheon, Portsmouth, RI**
Attendees	**Jianming Ye, Tom Silveria**

Titles of presentations that were made	
Meeting or Presentation 33	
Meeting Name	**Transition meeting with DDG 1000**
Purpose	**Discussion of replica communication problem and transitionable solution with DDG 1000 personnel**
Start Date	**8/30/2006**
End Date	**8/30/2006**
Location	**Telcon**
Attendees	**Joe Loyall, Matt Gillen, Paul Rubel, Andy Gokhale, Rick LaRowe, Graham Dooley**
Titles of presentations that were made	**Ideas and Technologies in Fault Tolerance for Transition to the PoR**
Meeting or Presentation 34	
Meeting Name	**NFD Telcons**
Purpose	**Discussion of transition of Node Failure Detection technology to DDG 1000**
Start Date	**8/4/2006**
End Date	**9/29/2006**
Location	**Telcon**

Attendees	**Matt Gillen, Allan Peckham, Kun Lu, Rich Lescroart**
Titles of presentations that were made	

References

1. Narasimhan, P., Dumitras, T., Paulos, A., Pertet, S., Reverte, C., Slember, J., Srivastava, D.: MEAD: Support for Real-time Fault-Tolerant CORBA. Concurrency and Computation: Practice and Experience (2005)

2. Object Management Group: Light Weight CORBA Component Model Revised Submission. OMG Document realtime/03-05-05 edn. (2003)

3. Cukier, M., Ren, J., Sabnis, C., Sanders, W., Bakken, D., Berman, M., Karr,D., Schantz, R.: AQuA: An Adaptive Architecture that provides Dependable Distributed Objects. In: Proc. of the IEEE Symposium on Reliable and

Distributed Systems (SRDS), West Lafayette, IN (1998) 245-253

4. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Gateways for Accessing Fault Tolerance Domains. In: Middleware 2000, LNCS 1795, New York, NY(2000) 88-103

5. Felber, P.: Lightweight Fault Tolerance in CORBA. In: Proc. of the International Conference on Distributed Objects and Applications, Rome, Italy (2001) 239-250

6. Vaysburd, A., Yajnik, S.: Exactly-once End-to-end Semantics in CORBA invocations across heterogeneous fault-tolerant ORBs. In: IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland (1999) 296-297

7. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-Tolerant Replication Based on Fragmented Objects. In: Proc. of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems- DAIS 2006. (2006) 256-271

8. Gill, C.D., Levine, D.L., Schmidt, D.C.: Towards Real-time Adaptive QoS Management in Middleware for Embedded Computing Systems. In: Proc. of the 4th Annual Workshop on High Performance Embedded Computing, Lexington, MA, MIT Lincoln Laboratory (2000)

9. Narasimhan, P.: Transparent Fault Tolerance for CORBA. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara (1999)

10. Marin, O., Bertier, M., Sens, P.: Darx - a framework for the fault-tolerant support of agent software. In: Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE 2003). (2003) 406-417

11. Object Management Group: Fault Tolerant CORBA Specification. OMG Document orbos/99-12-08 edn. (1999)

12. Baldoni, R., Marchetti, C., Virgillito, A.: Design of an Interoperable FT-CORBA Compliant Infrastructure. In: Proc. of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01). (2001)

13. Kemme, B., Patino-Martinez, M., Jimenez-Peris, R., Salas, J.: Exactly-once interaction in a multi-tier architecture. In: Proc. of the VLDB Workshop on Design, Implementation, and Deployment of Database Replication. (2005)

14. Amir, Y., Stanton, J.: The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Center for Networking and Distributed Systems, Johns Hopkins University (1998)

15. Dekel, E., Goft, G.: ITRA: Inter-Tier Relationship Architecture for End-to-end QoS. Journal of Supercomputing 28(1) (2004) 43-70

16. R. Kukura and T. Bracewell, "Raytheon company DARPA program composition for embedded systems (PCES) final report", June 30, 2005.

17. Schmidt, D.C., Levine, D.L., Mungee, S.: The Design and Performance of Real-time Object Request Brokers. Computer Communications 21(4) (April 1998) 294324

18. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document mars/03-05-08 edn. (July 2003)

19. Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France (November 2005)

20. Wang, N., Schmidt, D.C., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J.P., Schantz, R.E., Gill, C.D.: QoS-enabled Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2004) 131162

21. K. Dudziński and S. Walukiewicz. "Exact methods for the knapsack problem and its generalizations", European Journal of Operations Research, 28, pg 3-21, 1987

22. S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, editor, Surveys in Combinatorial Optimization, volume 31 of Annals of Discrete Mathematics, pages 213-258. North-Holland, 1987.